UNIT 1 TCP/IP PROGRAMMING CONCEPTS

Structure **Page Nos.**

1.0 Introduction 5

1.1 Objectives 5

1.2 Client Server Communication 6

1.2.1 Designing Client/Server Programs 7

1.2.2 Socket Concepts 11

1.2.3 IP Address and Ports 12

1.2.4 Byte Ordering 13

1.2.5 Sketch of Networking Connection 13

1.2.6 Active and Passive Sockets 14

1.3 Socket Fundamentals 15

1.4 Networking Example 18

1.5 Summary 20

1.6 Answers to Check Your Progress 20

1.7 Further Readings 22

1.0 INTRODUCTION

We are moving in the world of IT where information plays important role in our daily life. But exchange of information is also becoming important issue in today's world due to which most organisations in India and outside world use a substantial number of computers and communication tools. Computer networks (of an organisation or the Internet) allow the user to access programs and remote databases. The commonly used protocol suit, which provides these facilities, is called as TCP/IP. You might ask yourself, why should I learn about working of TCP/IP protocol or its programming? The answer is simple, as the Internet becomes more popular, more applications are developed. You should understand how TCP/IP these protocols work and also how you can develop your own application and protocols according to your need. At present TCP/IP has become an essential built-in component of most of the operating systems, including Unix, Windows 95, Windows 98, Windows NT, OS/2, and Linux etc. Before we can start writing network programs, we must understand how these systems (Internet or Intranet) actually communicate, which means that we have to learn a little about how the client and server communicate and work. First let's understand the client-server paradigm and next we will go in the details of socket and its structure which are used by network programs for communication.

1.1 OBJECTIVES

Our objective is to introduce you with basic concept of TCP/IP programming. On successful completion of this unit, you should be able to:

- have a reasonable understanding of the TCP/IP network architecture;
- describe the operation of simple client/server architecture;
- understand the role, meaning and structure of sockets;
- describe and understand how to use sockets; and
- demonstrate an understanding how to write client/server programs.

1.2 CLIENT SERVER COMMUNICATION

Computer Networks have revolutionised our lives. We have a lot of examples where we make use of computer network like, when we withdraw money from ATM machines or when we write e-mails or when we make computerized rail reservation (*Figure 1*).

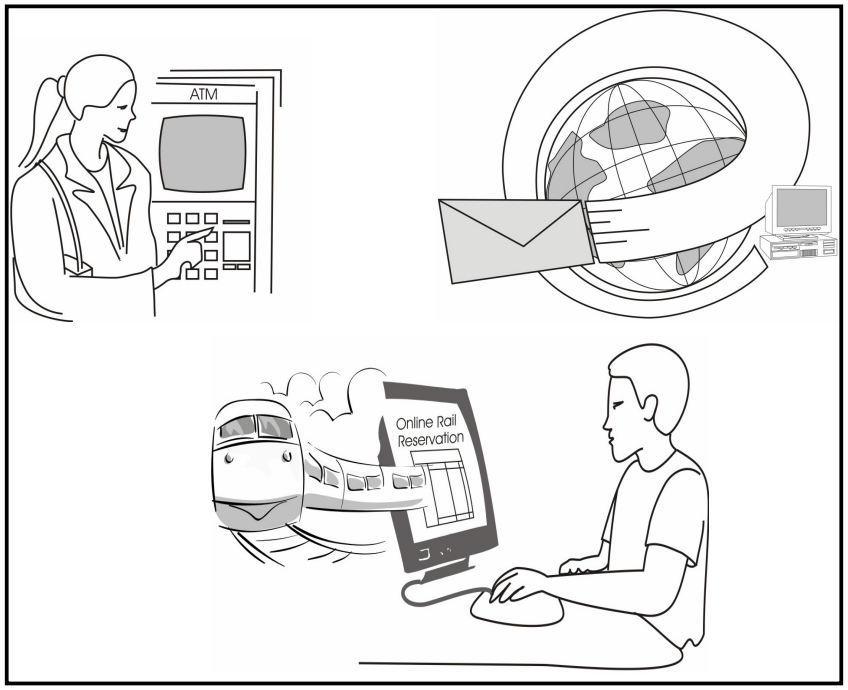


Figure 1: Use of Computer Network

All the above examples are nothing but computers communication. You must be wondering how these computers talk to each other and understand each other also? Because you are computer science students you may be thinking can I make them talk? Of course yes, this block is all about computer communication. But first of all try to understand how computers talk or network communication happens. Most of the computer networks use client/server model (*Figure 2*).

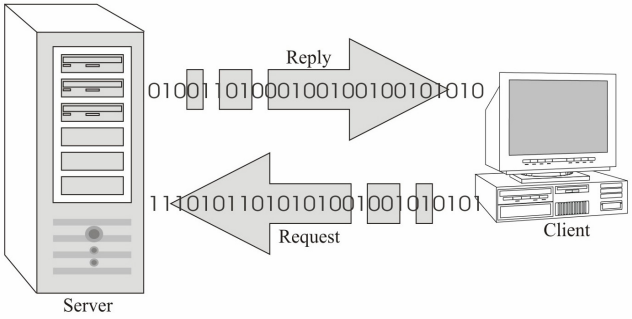


Figure 2: Client/server communications model

Client/server computing means one computer (or more) is acting as client computer one is behaving as serve which preferably of high configuration but can you think why we make a high configuration computer as a server? In simple words, client is defined as requester of services and server is defined as provider of services. Client and Servers are nothing but programs, which generally run on different machines/computers.

When client wants some information or wants to perform some task by server it has to contact the server. Therefore, the client initially needs the address of the server, and tries to contact the server using that address. Server is a program which is always running and waiting to serve clients. Server welcomes the client (if free) as the client contacts it. After contacting the server, the server welcomes that client (if free), the client informs about its own address to servers so that server can reply the client also. When both clients and servers exchange important information to each other connection establishes. Once with connection or link established, both client and server could exchange any information with each other. Lets take an analogy of doctor and patients to understand the client server paradigm in our daily life. Doctor can be considered as a server and patients as clients, patients want to get cured or want some information, so patient (s/he) should know the address of Doctor to whom s/he can contact. We assume doctor is always available for the patient, client inform about himself/herself. Now, doctor knows about patient details and history so they both can contact to each other according to their need.

We can say that a server is a program that is waiting for client so that it can do something for the clients and client program is defined as requester of services. Since server has to complete the task of many clients (may be thousands) that's why it should be of high configuration and have high speed, memory and disk space. Lets summarize the client/server communications.

A server is a process that is able to carry out some functions or services, like transferring files, translating host names to IP addresses, or any other task like finding inverse of a matrix.

1) Server is started on some computer system.
2) Client is started, either on same server computer or an different computer.
3) Client sends a request across the network to the server using server's network address which client should know before contacting.
4) Server listens the request and welcomes the client.
5) Client tells about its local address to server.
6) Connection established.
7) Both communicate to each other/servers serve the client like:
   - Print the files for the client,
   - Read or write a file present at the server etc.
8) When client's request is over and work is done, server goes back in waiting for other clients.

1.2.1 Designing of Client/Server Programs

Architecture of client and server program involves different issues, according to the system requirements. Here, in this section we have explained you about the characteristics and categories of client and server programs.

a) Characteristics of client program

Here we are explaining you about different characteristics of clients programs so that while writing programs you can always take care of these given characteristics. As you know, client program is an application program that becomes a client temporarily when remote access is needed.

- It is invoked directly by a user, and executes only for 1 session.
- Client program actively initiates contact with a server.

- Client program can access multiple services as needed, but actively contacts one remote server at a time.
- Client program does not require special hardware or sophisticated operating system.

b) Characteristics of server program

Server program is a special-purpose, program dedicated to provide generally one service, but server program can handle multiple remote clients at the same time. Server program has following characters:

- It is invoked automatically when a system boots, and continues to execute through many sessions. (inetd).
- Runs on a shared computer.
- Waits passively for contact request from arbitrary clients.
- Accepts contact from arbitrary clients, but offers a single service.

c) Iterative and concurrent programs

When server knows that client is going to take short amount of time, server program handles the client request one by one in linear manner. Such server are known as interactive or sequential server programs. But think, if server does not know how much time client will take, server may get busy with only one long request and other poor clients will keep waiting. In case server does not know about the time clients will take, servers typically handles it concurrent fashion, and such server is called concurrent server program. Most client software achieves concurrent operation because the underlying Operating System allows users to execute client programs concurrently or because user can execute client software simultaneously. A concurrent server invokes another process to handle each client request so that original server is always free to serve the clients. The program structure of iterative and concurrent server are given in the *Figure3*.

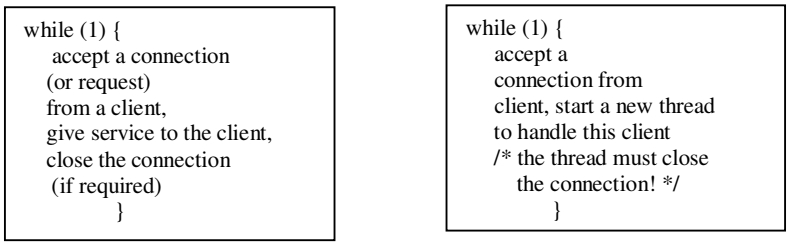


Figure 3: Program structure of interactive and concurrent servers

d) Standard and non-standard programs

Broadly client applications programs are categorised into standard and non-standard application programs. Standard application programs are those that use standard services like file transfer, electronic mail etc. of TCP/IP application layer. The application program which uses the services definitely privately or locally by the organisation according to its need is called non-standard application. For example, a site of private database system of some institute. Remember, standard application services are assigned to well known (universly reorganised) port number. Like file transfer protocol is university assigned 21port number. Non-standard applications are assigned to locally known (internally or privately recognised) port numbers. The site of Indira Gandhi National Open University is assigned to 3128 port number. System Administrator can assign these services to different port number according to its need and the options given by the authorities. It is important for network programmers to

understand the differences between these standard and non-standard services and their port numbers.

e) Connection and connectionless programs

When you design client/server program you must select the type of communication mode. You can either choose connection oriented communication mode or connection less communication mode, which directly refer to the transport layer protocols of TCP/IP. As you know TCP (Transmission Control Protocol) provides connection oriented services and UDP (User Datagram Protocol) provides connectionless services. If TCP is used between client & server, we can say it is connect oriented communication and if UDP then its connectionless communication. Connection oriented and Connection less services are distinguished in the scale of reliability-level. TCP provide reliability in communication between client and server, which means that whatever data is sent, the sender knows data is delivery properly or not. In connection-oriented communication underlying protocol (e.g., TCP) verifies the data delivery by acknowledgement and if not transmitted properly, data is automatically resent. It is also computes a checksum of data packets to provide the guarantee of non-corruption of data. TCP use the sequence numbers to maintain the proper sequencing and ordering of the data.

In contrast to this, connectionless services do not provide assurance about reliable delivery of data (no acknowledgement) or ensure that whether data will reach to the entitled process or not. For example, when client/server send some data to server it may be lost, resend, duplicated, can get corrupted, can be non-sequential or may be received or delivered out of order. But, instead of these problems and disadvantages the connectionless services (UDP) provides best-effort delivery. Retransmission is not performed but application must detect lost data and retransmit if needed. It is having very less overheads because of its simplicity; it provides very high transmission speed even in the wide area networks.

f) Stateful and Stateless programs

State information is the information that is maintained by a server about its states (some information between requests) of the ongoing interaction with client programs. If a server program is maintaining the states of ongoing communication it is called as stateful server program. FTP is the stateful server, otherwise it is stateless (like, HTTP server). Maintaining these states of the interaction improve the efficiency of the server. Maintaining small amount of information about the states reduce the size of message that client and server exchange. It is also helps the server to understand on different requests quickly; in addition to that it may also add some security features in the communication. In contrast the stateless server avoids the possibility of wrong or incorrectly reply, because sometime state information can become incorrect, duplicated, delivered out of order or client reboot between the communications.

Generally, server programs are designed into four categories accordingly to their capability of providing concurrency and their use of communication mode. These are summarized here in the *Figure 4*.

	Iterative	**Concurrent**
Connectionless	Iterative connectionless	Concurrent Connectionless
Connection-Oriented	Iterative connection-oriented	Concurrent Connection-oriented

Figure 4: Categories of server program

These four categories of server are further explained here:

i) **Concurrent Connectionless:** It is rarely used server design, where server create a new process to attend each request from the client, this concurrently provides the efficiency in the system but the time to create new process must be considerably less than the time required to compute the request. Generally, in this concurrent connectionless server design cost put for process creation become higher than the efficiency achieved with the concurrency. So, it is uncommonly used.

ii) **Concurrent Connection-oriented:** Most general design of server, it is suitable for service, which needs reliable transport of data in wide area networks and it can handle multiple clients simultaneously. It further has two different implementations:

- Concurrent process to handler connections.
- A single process and asynchronous I/O to handle multiple connections. (Process repeatedly waits for an I/O on any of the connections it has open and handles that request).

iii) **Iterative Connectionless:** Most commonly used server, this kind of server is suitable for services that need a less amount of processing for each request. Iterative servers are generally stateless, simple and less vulnerable to failures.

iv) **Iterative Connection-oriented:** It is less commonly used server. It is used for the services that need less amount of processing for each request but the reliability in communication is a necessary. But these kinds of servers waste their time in connection establishment and termination.

☞ Check Your Progress 1

1) Why the 'Server Computer' is high speed, memory and disk space?

……..……...

……

……..……...

……..……...

2) Write the difference between interactive and concurrent programs.

……..……...

……

……..……...

……..……...

3) List different categories of services.

……..……...

……

……..……...

……..……...

1.2.2 Socket Concepts

As we have studied earlier in the Unit 1 that socket is basically a combination of IP address and port number, but socket structure contains many more things. To understand in detail let study the concept of connections and association in this section.

a) Connections and Associations

As you know when client wants to communicate to server it first establishes "Connection". Here connection is nothing but a communication link between client program and server program. As we have defined in our present section, when connection is established between client and server, they exchange a set of important information that is known as "Association". Lets see what exactly we mean by association.

The term association is used for the 5-tuple's that completely specify the two processes that comprise a connection.

Association {
protocol,
local address,
local process,
remote address,
remote process.
}

Let's understand the 'Association' with the help of one example. Machine 'A' is local host and wants to establish connection with Machine 'B' which is a remote host. Now Machine 'A' has to define which protocol it will use to establish connection like TCP or UDP (in-case of TCP/IP). Remote and local address of machines, are like 'IP address' of machine in TCP/IP, which actually identify the location of machine in network.

Local Process and Remote Process are used to identify the specific process in system that is involved in establishing connection. For example, **Association** needed for connection in TCP/IP is (UDP, 192.23.15.10, 1211, 192.64.10.12, 2102) where UDP is a protocol. The association with 5-tuple's is also known as full association but we also have **half association.**

Like:

{protocol, local address, local process}
&
{protocol, remote address, remote process}

Which define each half of the connection. This half Association is also called as '**Socket**' or '**transport address**'. The term Socket's popularized by Berkley Unix networking system. In the next section we will discuss Socket in detail.

b) What is Socket?

The basic building block for computer communication is the *socket*, which is an end-point of communication. Sockets allow communication between two different processes on the same or different machines. Let's see the concept of sockets in general term. In term of electricity we use "Electrical socket" which use to get electrical power plug an appliance's power cord into a socket in the wall. When you want to talk to your friend through telephone, you can connect your telephone with different end-points (Telephone-Socket) available in your home as shown in the

Figure 5. The same concept we can realize when we talk about the Socket for communication between machines.

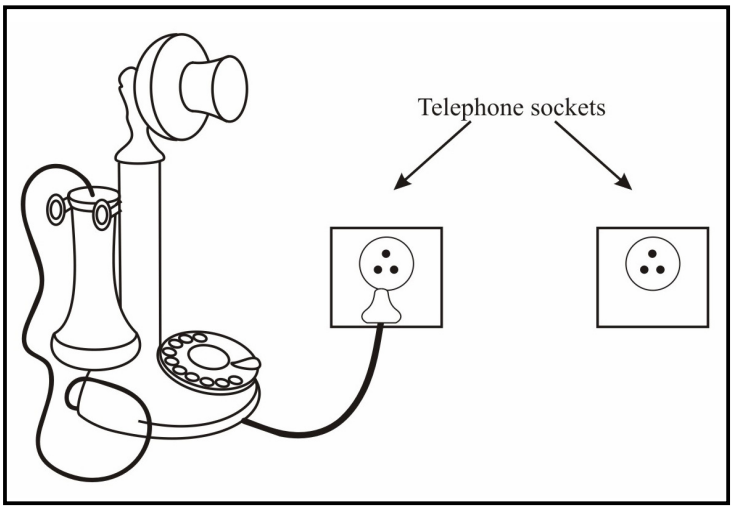


Figure 5: Socket is an end-point of communication

Why is it called a socket? Because of the analogy to plugging a wire into something that can understand what is carried by the wire. When two applications want to talk to each other so, one application tells the OS to open a socket and through socket it uses communication protocol, at receiver side one socket will be ready with the open socket. Once connection is established application can send and receive data.

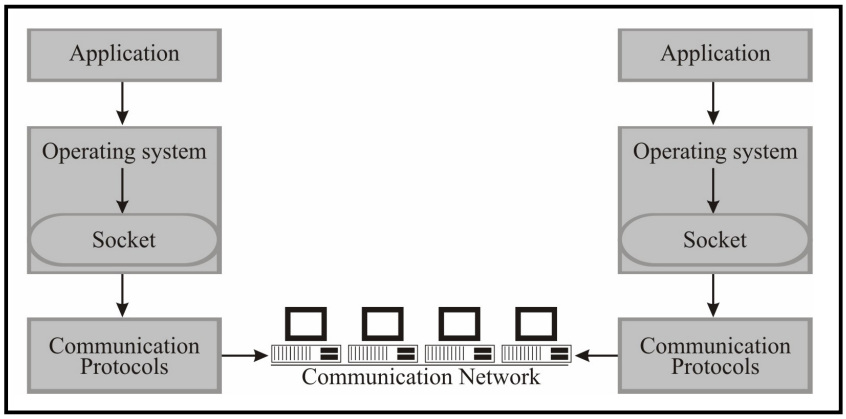


Figure 6: Sockets are basic building block of communication

1.2.3 IP Address and Port

An internet address or IP address is number made of 32 bit or 4 bytes (number between 0 and 255) written in the form of 'dotted notation' for example "192.168.10.10" **is a valid IP address** usually one computer has only one IP address but a computer might have more than one such address, if computer has more than one NIC (Network Interface Card) connected to internet. Therefore, when we mean host we should understand, IP address that identifies a host Interface not a host but usually we mean it to host.

When we talk about Network communication we say IP address identifies the host computer. However, usually more than one application program running on a computer use the network. Because of this reason TCP/IP use another level of address called port number, which identify the process running on computer.

Let's see an example, you want to call me, you know the IGNOU phone number is 29536207 but in IGNOU we have many teachers that is why you have to use my extension 2229, here in analogy to network communication, telephone number is IP address and extension number is as Port Number. Explain number is used, as another level of address is to resolve among teachers in IGNOU.

In networking, communication address is made up of an IP address and a Port Number. Port Number is a 16-bit identifier and each host has 65336 ports. In TCP/IP some ports are reserved for specific application for example, port number 21 is reserved for FTP, Port number 23 is for telnet, 80 for HTTP.

Name	Port Range	Use
The Well Known Ports	0 to 1023	Reserved for common services like telnet, ftp.
The Registered Ports	1024 to 49151	Registered for companies and organisation
The Dynamic and/or Private Ports	49152 to 65535	Available for the rest of world to use.

1.2.4 Byte Ordering

Some computers are "big endian". This refers to the representation of objects such as integers within a word. In big endian machine, the high byte of an integer is stored in low byte address. In little endian, the high byte of an integer is stored in high byte address as shown in *Figure 7*. A Sun Sparc is big endian. So the number 2 + 3 * 256 would be stored as:

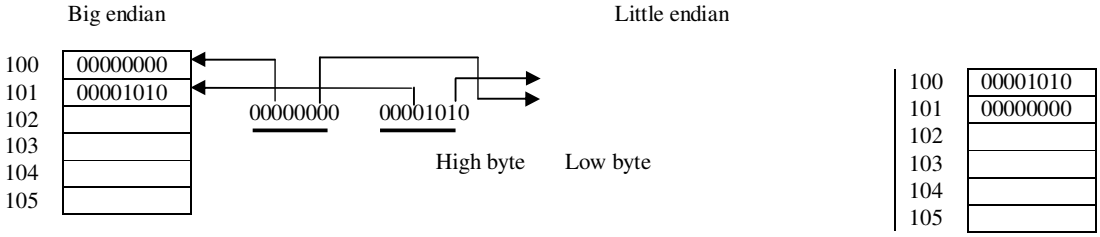


Figure 7: Data representation in big and little endian

		3	2
3	2	1	0

A "little endian" machine stores them the other way. The 386 is little endian.

2	3		
3	2	1	0

If a Sparc sends an integer to a 386, what happens? The 386 sees a data "2 + 3* 256" as "2 *16777216 + 3 * 65536".

1.2.5 Sketch of Networking Connection

The client/server model is used to divide the work of networks into two parts. One part knows how to do a certain task or how to provide certain service, this part is known as server program. Other part defines how to talk to user and how to connect to server program, known as client program. Server may give service to many different clients either simultaneously (concurrent services) or one after the other (Iterative services). On the other hand client talks to single user at a time. In network

program we can have different possibilities depending on the needs. But first of all we will discuss only simple one. Let's see the following example where one client talks to one server. The main feature of client is to provide convenient user interface, hiding the details of how the server communicates from the user. The client needs to first establish a connection with server, on given address. Generally, client does the following steps:

1) Create a socket.
2) Specify the address and service port of the server program.
3) Establish the connection with the server.
4) Send and receive information.
5) Close the socket when finished, terminating the conversation.

To establish and complete a connection the server-side would follow these steps:

1) Create a socket.
2) Listen for incoming connection requests from clients.
3) Accept the client connection.
4) Send and receive information.
5) Close the socket when finished, terminating the conversation.

1.2.6 Active and Passive Socket

The client application always starts the communication. It creates a socket and actively attempts to connect to the server program. This is known as active open or active socket. On the other hand, the server application creates a socket and passively listens for incoming connection requests from clients. This socket is passive open, that's why this socket is called as passive socket. This is also same as the telephone system where one phone is always waiting for incoming calls.

When client initiates a connection, the server notices that same process is trying to connect with it. By accepting the connection the server establishes a logical communication path between two programs. (Generally, in practical situation, the act of accepting a connection creates a new socket. The original socket remains unchanged so that it can continue its availability to clients for additional connections when the server no longer wishes to listen for connection).

☞ Check Your Progress 2

1) What are the types in association?

   ……...
   ……
   ……

2) What is a Socket?

   ……...
   ……
   ……

3) Write the difference between active and passive socket.

   ……...
   ……
   ……

1.3 SOCKET FUNDAMENTALS

To standardize network programming, Application Programs Interface's (API's) have been developed. An API is a set of declarations, definitions and procedure's followed by programmers to write clients server programs. For network programs commonly developed API's are Socket Interfaces, Transport Level Interface (TLI), Stream Interface, Thread Interface and Remote Procedural Call (RPC). Let's discuss socket interface in details but first see the data types defined in socket interface.

There are u_char u_short and u_long data type.

```
u_char     // character unsigned 8 bit
u_short   // short integer  unsigned 16 bit
u_long   // long integer unsigned  32 bit integers
```

From application point of view, the differences between network protocols are address schemes used. For TCP/IP, an ideal API would be one that understand IP addressing and port numbers. Since the socket library is designed to be used for multiple protocols, addresses are referenced by a generic structure as follows:

```
struct sockaddr {
u_char   sin_len;        /* length of structure */
u_short sa_family;     /* address family: AF_XXX value */
char sa_data[14];      /* up to 14 byte of protocol specific address*/
                    };
```

Originally sa_len was not there in the structure. The **sa_family** field specifies the type of protocol. But in case of TCP/IP, this field is always set to AF_INET (where INET stand for Internet). The remaining 14 bytes (sa_data) of this structure are always protocol dependent for TCP/IP, IP addresses and port numbers are placed in this field. The contents of the 14 bytes of protocol dependent addresses are interpreted according to the type of address. (In TCP/IP this is defined in < netinet/in.h >header file.)

Internet socket address structure: The structure *sockaddr_in* describing an address in the Internet domain is defined in the file <netinet/in.h>. The application program that use the TCP/IP need a specific type of socket address structure, which can hold IP address, port number and protocol family. This structure is called as sockaddr_in., which have five fields.

```
struct sockaddr_in
     {
u_char        sin_len;        /* length of structure  */
u_short     sin_family;      /* for Internet it is AF_INET */
u_short     sin_port;        /* 16-bit port number */
struct in_addr  sin_addr;    /* 32-bit address */
char           sin_zero[8];  /* unused, reserved for future */
     };
```

In this structure first and last field are normally not used. A sockaddr_in structure contains an in_addr structure as a member field. In_addr has the following structure.

```
struct in_addr {
u_long   s_addr;          /* 32-bit address */
                  };
```

Note one thing here about the **sock_addr** & **sockddr_in** structures; these both structures are compatible with each other. Both structures are 16 bytes in size and first two bytes of both structures are the family field. Thus, struct sockaddr_in can always be cast to struct sock_addr. It is important to remember that these fields always need to be set and interpreted in network byte order (this topic we will discuss in unit 3 of this Block).

A socket is defined in Operating System as structure, which has five fields. (Remember the five tuple's in association, same here in the structure of socket). Let's see these fields of structure in the *Figure 8*:

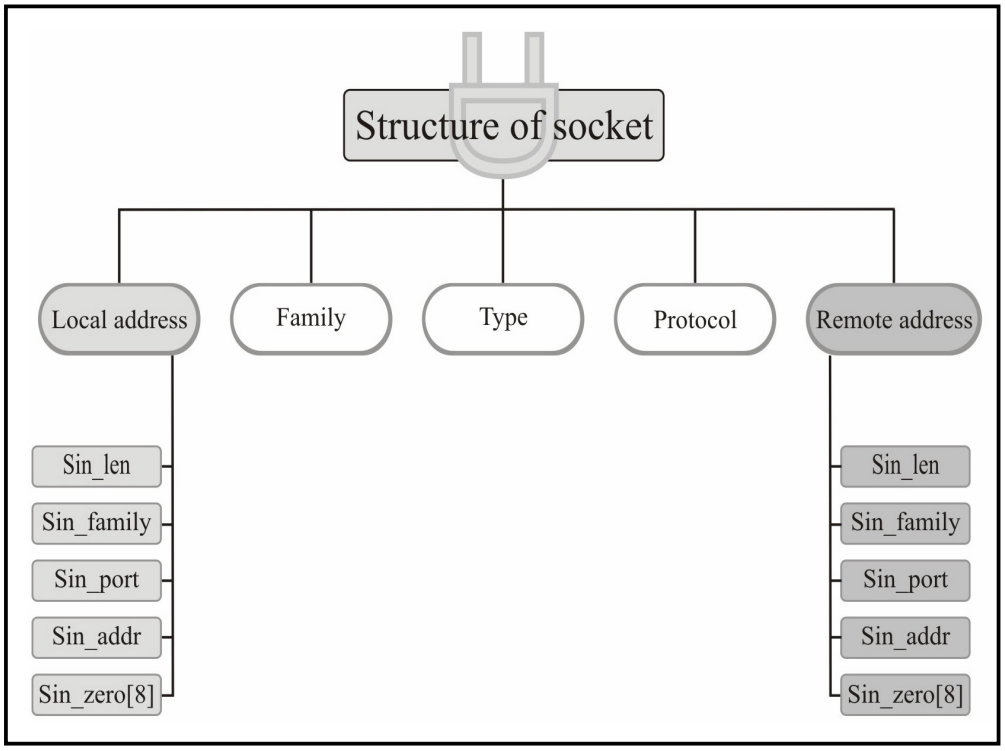


Figure 8: Structure of socket

1) **Family:** An application program specifying a socket family defines the protocol group needed for communication. It can be Unix domain protocols, Internet domain protocol, Xerox NS protocols and IMP link layer (IMP is Interface Messaging Protocol) as given below. In case of TCP/IP this field is AF_INET.

2) **Type:** An application program specifying a socket type, which indicates the desired communication style for the socket. This field defines the type of socket like stream socket, data-gram socket and raw socket. Each socket has a type associated with it. The socket type determines the socket communication properties like reliability, ordering, and prevention of duplication of messages. The basic set of socket types is defined in the **sys/socket.h** header file. Total we

have five types of Socket options available. Let's discuss in details what are these types of sockets:

i) **Stream Socket:** Stream socket is used with connection oriented protocol. It provides stream model of communication. A *stream* socket provides for the bi-directional, reliable, sequenced, and unduplicated flow of data without record boundaries. This service is reliable and connection oriented. Transport protocols support this socket type and ensure that the delivery is in-order, without loss or error or duplication, if not, then abort the connection and report the error to the user. Message boundaries are not present in stream sockets. It can be used in both Unix and Internet domain. In case of Internet domain it uses TCP, which is connection-oriented protocol.

ii) **Datagram Socket:** Datagram sockets are used with connection-less protocols. It provides datagram model of communication. It supports bi-directional flow of data, between sender and receiver and data is not promised to be in sequence, reliable, or unduplicated. That's why in case of datagram socket, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. This type of socket is generally used for short messages, such as a name server or timeserver, since the order and reliability of message delivery is not guaranteed.

iii) **Raw Socket:** Raw Socket, as its name indicates, provides access to internal network protocols and interfaces. A raw socket allows an application direct access to lower-level communication protocols (like, IP layer in case of TCP/IP). Raw sockets are developed for advanced programmers who want to build new protocols or who are interested to use the low level feature, that are not directly accessible through a normal interface. Raw sockets are normally datagram-based, though their exact characteristics are dependent on the interface provided by the protocol.

iv) **Sequenced Packet Socket:** It provides sequenced, reliable, and unduplicated flow of information. A sequenced packet socket is similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as part of the NS socket abstraction, and is very important in most serious NS applications.

v) **Reliably Delivered Message Socket:** The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. There is currently no support for this type of socket, but a reliably delivered message protocol similar to xerox's Packet Exchange Protocol (PEX) may be simulated at the user level.

3) **Protocol:** The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist; in that case a particular protocol must be specified.

4) **Local Socket Address:** A pointer to a buffer of type struct sockaddr_in that contains the local address to which the socket is to be bound.

5) **Remote Socket Address:** A pointer to a buffer of type struct sockaddr_in that contains the remote address to which the socket is to be bound.

Now lets summarise the characteristics of socket:

- A socket is referenced by an integer. That integer is called a socket descriptor.
- A socket exists as long as the process maintains an open link to the socket.

- You can name a socket and use it to communicate with other sockets.
- Sockets perform the communication when the server accepts connections from them, or when it exchanges messages with them.
- You can create sockets in pairs (only for sockets in the AF_UNIX address family).

1.4 NETWORKING EXAMPLE

As we have discussed earlier that client program can use the connection oriented reliable service or connection less unreliable services. So you should know the algorithms or steps for these client program.

Connection oriented and Concurrent client

Here we have given you an algorithm, where client opens the socket, sends a request and gets the response.

Step 1: Identify the IP address and port No. of a server.

Step 2: Open a socket.
Step 3: Choose TCP protocol (a connection oriented protocol) and an unused protocol port.

Step 4: Connect Socket to the server.

Step 5: Using application level protocol, communicate with server.

Step 6: Close the connection, when the job is over.

Connectionless Client

Step 1: Same as step 1 in connection oriented client

Step 2: Same as step 2 in connection oriented client

Step 3: Choose UDP Protocol (a connectionless protocol)

Step 4: Inform the server about the client address so that server can communicate the client.

Step 5: Same as step 5 connection oriented client

Step 6: Same as step 6 connection oriented client

Now you learnt the client program algorithm, but without server these are not useful so you must learn how to design different kinds of server programs. As we have early classified them into 4 types in the section, let's make there algorithms one by one.

Iterative and Connection oriented server

Step 1: Create a socket, and associate it with the well-known address (which is known to the client).

Step 2: Make the socket in passive mode.

Step 3: Welcome the connection request from the client, and create a new socket for the connection

Step 4: Read request from client repeatedly and give them reply according to application protocol.

Step 5: When the job of particular client is over close that connect and rather step 3.

Here at a time, a single process handles one connection from client, only one when it finish its request than only it accepts next connect request.

Iterative and connectionless server

Step 1: Same as step 1 iterative and communication Server

Step 2: Same as step 2 iterative and communication Server

Step 3: Repeatedly read the next request from the client and send the reply according to the protocol.

Connectionless and concurrent server

Master 1: Create socket and fix it to some address (well known to client) offered Socket must be left unconnected.

Master 2: Receive next request from client and create copy of the same (child process).

Slave 1: Receive the specific request

Slave 2: Reply accordingly to application protocol and give response to client

Slave 3: Terminate the child process after completion of the task.

Connection oriented concurrent server

Master 1: Create socket and fix it to the address, which is known to the client. Socket must be unconnected.

Master 2: Make the socket as passive socket

Master 3: Continuously receive the connection request from client and create slave server process to handle the communication with client.

☞ Check Your Progress 3

1) What are the data types defined by the socket interface?

……..……...

……

……

……

……

2) Write the structure of socket address.

……..……...

……

……

……

……

3) Write the algorithm for interactive connection oriented server.

……..……...

……

……

……

……

1.5 SUMMARY

In this unit we have discussed different issues related to client server communication so that you will have good understanding of different servers and their services like connection oriented and connectionless. The procedure of basic client and server was also explained in detail. After completing the theoretical conceptualization of client server communication we have defined the socket, which is considered as an end point of communication in network programming. The different components related to the sockets like, IP address, port, address family and socket types are also explained in this unit. The concept of byte ordering and its use in networking is explained briefly with the help of an example which you will again study in detail during the unit socket programming. Further the differences between Active and Passive Socket was explained and at the end of the unit we have given you some algorithms and procedure of different client server communication. In the next unit *Socket Interface* we have given basic socket calls, which will provide you the knowledge of the functions and their characteristics.

1.6 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) Server is a program that is waiting for client so that server can do something for the client and client program is defined as requester of services. Because server has to complete the task of many clients (may be thousands) that's why it should be of high configuration and have high speed, memory and disk space.

2) When server knows that client is going to take short amount of time then server program handles the client request one by one in linear manner, which is known as interactive or sequential server. In this, if server don't know how much time client will take, server will get busy with one request itself and other poor clients will wait. A concurrent server invokes another process to handle each client request so that original server can always be free to serve client. That's why we can say practically it is more efficient.

3) Server programs are designed into four categories accordingly to their capability of providing concurrency and their use of communication mode.

   These are following:

   1) Concurrent Connectionless
   2) Concurrent Connection-oriented.
   3) Interactive Connectionless
   4) Interactive Connection-oriented

Check Your Progress 2

1) The term association is used for the 5-tuple's that completely specify the two processes that comprise a connection.

```
Association {
        Protocol,
        local Address,
        local process ,
        remote address,
        remote process.
    }
```

2) The basic building block for computer communication is the *socket*, which is an end-point of communication. Sockets allow communication between two different processes on the same or different machines.

3) The client application always starts the communication. It creates a socket and actively attempts to connect a server program. This is known as active open or active socket. On the other hand the server application creates a socket and passively listens for incoming connection from clients that is passive open situation. That's why this socket is called as passive socket.

Check Your Progress 3

1) The main dates types defined in this socket interface are u_chart u_shorts and u_long data type, it's meaning are defined below:

- u_chart : character unsigned 8 bit
- u_shorts : short integer unsigned 16 bit
- u_long : long integer unsigned 32 bit integers

2) The general socket address is given below:

```
struct sockaddr {
        u_char   sin_len;        /* length of structure */

        u_short sa_family; /* address family: AF_XXX value */

        char sa_data[14]; /* up to 14 byte of protocol specific address*/

        }
```

Where sa_len indicates the length of the address structure. The sa_family field specifies the type of protocol. The remaining 14 bytes (sa_data) of this structure are always protocol dependent for TCP/IP, IP addresses and port numbers are placed in this field. The contents of the 14 bytes of protocol dependent addresses are interpreted according to type of address.

3) The algorithm for Interactive and Connection oriented server is as given below:

Step 1: Create a socket, and associate it with the well-known address (which is known to the client).

Step 2: Make the socket in passive mode.

Step 3: Welcome the connection request from the client, and create a new socket for the connection.

Step 4: Read request from client repeatedly and give them reply according to application protocol.

Step 5: When the job of particular client is over close that connect and rather step 3.

Here at a time, a single process handles one connection from client, only one when it finishes its request than only it accepts next connect request.

1.7 FURTHER READINGS

1) Andrew S. Tanenbaum, *Computer Networks*, Third edition.

2) Behrouz A. Forouzan, *Data Communications and Networking*, Third edition.

3) Douglas E. Comer, *Internetworking with TCP/IP Vol.1*: Principles, Protocols, and Architecture (4th Edition).

4) William Stallings, *Data and Computer Communications*, Seventh Edition.

5) W. Richard Stevens, "*UNIX Network Programming*", Prentice Hall.

UNIT 2 SOCKET INTERFACE

Structure **Page Nos.**

2.0 Introduction 23

2.1 Objectives 23

2.2 Elementary Socket System Calls 23

2.2.1 Socket System Call 24

2.2.2 Bind System Call 27

2.2.3 Connect System Call 28

2.2.4 Listen System Call 30

2.2.5 Accept System Call 31

2.3 Elementary Data Transfer Calls 34

2.4 Closing a Socket 38

2.5 TCP and UDP Architectures 39

2.6 Networking Example 41

2.7 Summary 43

2.8 Answers to Check Your Progress 44

2.9 Further Readings 45

2.0 INTRODUCTION

Socket is considered as an end-point, which is used by a process for bi-directional communication in which another socket is associated with another process. As we discussed earlier a socket is like, a file descriptor available in Unix for file communication (I/O) like, open (), create (), close (), read () and write (). Similarly for network communication (I/O) socket behaves like, socket descriptor by which we can read () write () data. File is like a sequence of characters which we can read using repeated read operation in connection oriented mode and in connectionless mode we have to get whole message in a single read operation. But because of the complexity and the requirements of network communication we need many more system calls that we will discuss in this unit.

2.1 OBJECTIVES

Our objective is to introduce you, with the elementary socket calls. After successful completion of this unit, you should be able to:

- have a reasonable understanding of the Elementary System Calls;
- understand the use of basic data transfer calls;
- describe the TCP and UDP Client/Server Architecture; and
- demonstrate an understanding the simple network programs.

2.2 ELEMENTARY SOCKET SYSTEM CALLS

As we discussed in unit 1 when a socket is created, it is allocated a slot in the file descriptor table exactly the same way a file is. Once this socket is associated with a file, we can do I/O such as read and write. Recall that generally sockets are for network communication and network I/O deals with a completely different set of problems than file I/O, but wherever possible, actions corresponding to file I/O are

accomplished. But what is the difference between file I/O and networking I/O? let's find out:

- The client-server relationship is not symmetrical. The client and server processes must be assigned the responsibilities.
- Connectionless protocols do not have an open command because any operation could come from any process.
- Peer process names and file names are important for authority uses.
- More parameters must be specified (protocol, local address, local post server address, server process).
- All computers do not share the same data format (e.g., little endian versus big endian). For details see unit 3.

We now describe the elementary system calls which require to develop Network programs.

2.2.1 Socket System Call

In the early 1980s, with ARPA project, University of California at Berkeley had the responsibility to transport the TCP/IP protocol suit to Unix operating system. It was decided to use Unix system calls with addition to new system calls, if required, as the result new socket interface developed, which become popular as Berkeley UNX or BSD (Berkeley Software Distribution) version 4.1. We are going to discuss BSD Unix system calls with you in this section.

The socket system calls is used by any process to create a socket for doing any network I/O. The structure of socket, we have already discussed as general but here we will discuss in detail with programming concept. The structure of this is given below.

#include <sys/types.h>
#include <sys/socket.h>

int socket (into family, int type, int protocol);

Condition	Returns
Successfully	**Small integer value called the socket descriptor**
Unsuccessful	**Error (-1)**

Socket function creates a socket, it sets values for only family, type and protocol of socket structure the other fields are set by other functions or by operating system. If socket system call creates socket successfully then it returns small integer value called the socket descriptor sockfd (which is similar to a file descriptor) which uniquely identifies the socket. If socket is not created it means there is an error and it returns –1.This socket descriptor sockfd, is used by other functions to refer the socket. (see the socket description tables and its contents in the *Figure1*).

Whenever you will use socket system call you should include both of these header files sys/types.

#include<sys/types.h>: This header file contains definitions of a number of data types used in system calls.

#include<sys/socket.h>: The header file socket.h includes a number of definitions of structures needed for sockets.

Socket header files contain data definitions, structures, constants, macros, and options used by socket subroutines. An application program must include appropriate header files to make use of structures or other information a particular socket subroutine requires. Some other header files are also important for Unix socket programmers: These are given in *Table 1*. You should remember that header files required for some particular calls or API may differ from system to system. So you should always check it with the online manual available in Unix.

Table1: Socket Header Files

Socket Header File	Description
inet.h	takes the place of both in.h and inet.h header files. It defines values common to the Internet.
ip.h	defines values used by the Inter-network Protocol (IP) and details the format of an IP header and options associated with IP.
netdb.h	defines the structures used by the "get" services.
errno.h	defines the errors that can be returned by the socket library when requests are made to it.
sockcfg.h	describes the socket configuration structure
socket.h	defines most of the variables and structures required to interface properly to the socket library.
sockvar.h	is needed when compiling the socket configuration file
tcp.h	describes those options that may be set for a socket of type SOCK_STREAM.
uio.h	describes the structures necessary to use vectored buffering of the socket library.

Socket Descriptor

As you know, in Unix, if some application needs to perform input/output function, it calls the "open" function to create the file descriptor which has further access to the file. Unix uses descriptor as an index into process descriptor table, which follows the pointers to the data structure that holds all details about file. Similarly, when some application calls "socket" the operating system allocates a new data structure as shown in Figure 1(a) and 1(b) to hold the details required for communication and enter the pointer into the description table.

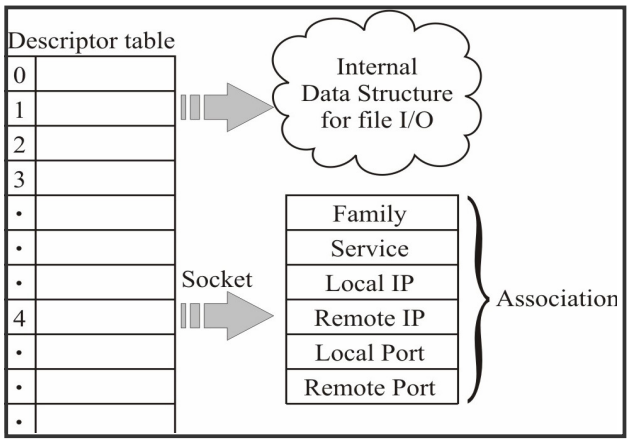


Figure 1 (a)

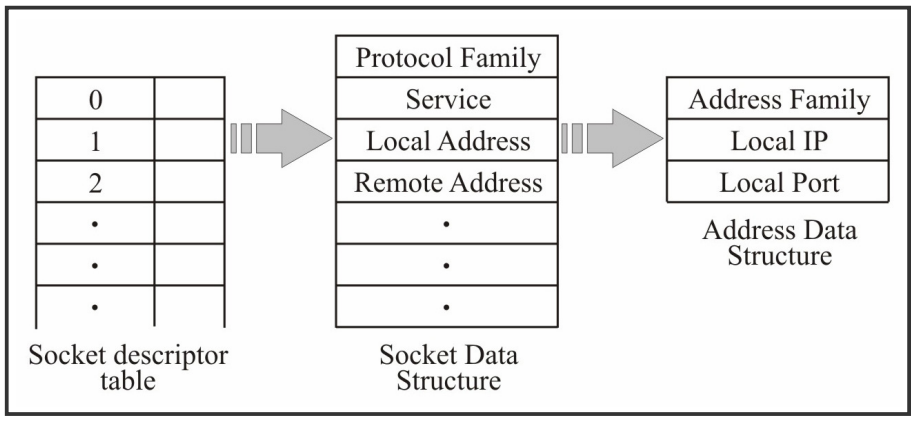


Figure 1 (b)

Figure 1: Socket Descriptor

Socket descriptor defines an address family to provide the freedom to different protocol families for choosing their own representation for their address. A single protocol family can use more than one address families to define address representation. TCP/IP protocol family uses a single address representation; this address family is symbolically represented by AF_INET (Address family internet). *Don't get confused with the PF-NET, both denote the address family in TCP/IP.*

Socket family: As you know socket family defines the protocol group needed for communication. At the time of programming you should choose one of the given options but because we are concerned about TCP/IP you need to remember AF_INET.

```
AF_UNIX /*Unix internal protocols */
AF_INET /* internet protocols */
AF_NS /* Xerox NS protocols */
AF_IMPLINK /* IMP link layer */
```

Socket types: The Type parameter in socket system call specifies the semantics of communication. Sockets are typed according to the communication properties visible to a user. Mostly Processes can communicate only between sockets of the similar type. If underlying communication protocols support, communication between different types of sockets can happen. As we have discussed earlier also you have following options available with "socket",

/*Standard socket types available*/

```
#define  SOCK_STREAM          /* stream socket */
#define  SOCK_DGRAM           /*datagram socket*/
#define  SOCK_RAW             /*raw socket*/
#define  SOCK_SEQPACKET       /*sequence packet socket */
#define  SOCK_RDM             /*reliably-delivered message*/
```

Protocol: The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family.

Protocol is dependent on the services we are using in our communication. For example, if we are using Stream Socket type we must use a protocol that provide a connection-oriented service, like, TCP. But if we are using Datagram Socket type

which provide connectionless services then available protocol is UDP. Please see the Table 2 given below):

Table 2: Protocol corresponding to socket type and socket family

	AF_UNIX	AF_INET	AF_NS
SOCK_STREAM	YES	TCP	SPP
SOCK_DGRAM	YES	UDP	IDP
SOCK_RAW	NOT DEFINED	IP	YES
SOCK_RDM	NOT DEFINED	NOT DEFINED	SPP

All combinations of socket family and type are not valid. The *Table 2* (reference from Unix Network Programming by Richard Stevens) server shows the valid combinations along with the actual protocol that is selected by pair.

To create a TCP socket:

```
int sockfd;
sockfd = socket (AF_INET,
SOCK_STREAM, 0);
```

To create a UDP socket:

```
int Sockfd;
sockfd = socket(AF_INET,
SOCK_DGRAM, 0);
```

2.2.2 Bind System Call

After creating "socket" and before Sending/Receiving data using socket, it must be associated with a local port and a network interface address. That's why we can say mapping of a socket to a port number and IP address is called a "binding". Why binding is needed, as you know server use socket so it can attend client request on specific port. Socket must be associated with particular port on one network interface. But think when we can have multiple network addresses on one host and each network, we have some port addresses. In this case with the help of bind we can define, for which network address (IP address + port address) with which port is associated, otherwise imagine how much confusion can happen.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr_in *localaddr, int addrlen);
```

Here in the above prototype of bind, sockfd is File descriptor of local socket, as created by the socket function. Localaddr is a pointer to protocol address structure of local socket. The special address ANY_ADDR can be used to allow the connection to be made on any of the host's interfaces and addrlen is indicating length in bytes of structure referenced by address. On success, bind () returns a zero. On failure, it returns -1 with an error number.

Use of Bind

During networking programming we find that there are three user of Bind system call.

- A specific network address and port address can be registered to a client.
- Server needs to register a specific network & port address with its socket, basically it informs the system that "The address associated with me is this, and if you get any message on this address just transfer it to me".

- In case of connectionless client (see your unit 1 of block 1 to know about connectionless and connection oriented services), it needs to assure that system has provided some valid unique addresses, so that when server sends some message it will reach the desired client. This approach of client is same like us, when we write letter we always check whether we have written own address on that envelop or not so that we can get reply on that address.

The bind system call fills the two tuple of association, Local address and Local process elements. When we will use Bind function the client needs to call socket system call because it needs to use the returned value as socket descriptor.

When binding is over, then in-case of UDP socket it is ready to send and receive datagram's. For TCP sockets, the socket is ready to connect or accept calls. Let's see what are these accept and connect call.

Let's have an example:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#define CLIENTPORT 8090

main()
{
   int sockfd;
   struct sockaddr_in local_addr;

   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   client_addr.sin_family = AF_INET;     /* host byte order */
   client_addr.sin_port = htons(CLIENTPORT); /* short, network byte order */
   client_addr.sin_addr.s_addr = inet_addr("192.10.10.10");
   bzero(&(client_addr.sin_zero), 8);    /* zero the rest of the struct */

   /* error checking for bind(): */
   bind(sockfd, (struct sockaddr *)&client_addr, sizeof(struct sockaddr));
   .
   .
   .
  }
```

In socket programming we have different methods which may be useful to you, when you want to choose an unused port at random you can write /* choose an unused port at random */

and for choosing IP address you can use /* use client IP address */

2.2.3 Connect System Call

Generally "connect" function is used by client program to establish a connection to a remote machine (server).

Syntax of connect is given below:

#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr_in *server_addr, int serv_addrlen);

connect() returns 0 if successful or returns -1 on unsuccessful and sets errno

In the connect call, sockfd is a socket file descriptor returned by socket(), server_addr points to a structure with *destination* port and IP address and serv_addrlen set to sizeof (struct sockaddr).

Initial code necessary for a TCP client:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_IP   "190.20.10.12"
#define SERVER_PORT 1729

main()
{
 int sockfd;
 struct sockaddr_in ser_addr;   /* will hold the destination addr */

 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0))<0)
 {
  perror("socket error ");
  exit(1);
 }

 Ser_addr.sin_family      = AF_INET;           /* inet address family */
 Ser_addr.sin_port        = htons(SERVER_PORT);   /* destination port */
 Ser_addr.sin_addr.s_addr = inet_addr(SERVER_IP); /* destination IP address */
 memset(&(dest_addr.sin_zero), '\0', 8);   /* zero the rest of the struct */
 /* In function void *memset(void *s, int c, size_t n); The  memset() function
fills the first n bytes of the memory area pointed to be s with the constant byte c.*/

 if (connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr))<0)
 {
  perror("connect error");
  exit(1);
 }
```

During the bind call example earlier we have ignored the local port number, now we are concerned about the remote port. The kernel will choose a local port, and the site we connect to, will automatically get this information from client.

If connectionless client (UDP) use the connect (), then the system call just stores the server address specified by the process, so that the system knows where to send any future data the process writes to the sockfd descriptor. Also, the socket will receive only datagrams from this address. Note that the destination address is not necessary for each Datagram sent, if we are connecting a UDP socket.

Connect system call result in actual connection establishment between the local and remote system in case of connection-oriented protocol. At this point some agreement for the future data exchange happens like buffer size, amount of data, acknowledgement etc., as we have discussed earlier in Unit 3 of Block 1. This exchange of information between client and server are known as 3-way handshake,

To use connect function, first of all client needs to call the socket (), then connect () function sets value of server socket address and client socket address is either provided by bind system call or set by the operating system.

2.2.4 Listen System Call

TCP server, binds to a well-known port and waits for a client to try to connect to it. This process is called as "listening" and it is performed by calling the listen () system call.

The Listen system call is used in the server in the case of connection-oriented communication to prepare a socket to accept messages from clients. It creates a passive socket (recall the concept of passive and active mode of socket) from an unconnected socket. 'Listen' initializes a queue for waiting connections. Before calling listen, socket must be created and its address field must be set.

Usually Listen() is executed after both socket() and bind() calls and before accept() [accept system call will be discussed in next sections].

#include<sys/types.h>
#include<sys/socket.h>

int listen(int sockfd, int Max_conn)

On success, listen() returns "0"
On failure, it returns "-1" and the error is in errno.

Listen takes two parameters, first the socket you would like to listen on and another is the Maximum number of request that will be accepted on the socket at any given time. Socket you would like to listen on is represented here as **sockfd** is the usual socket file descriptor from the socket() system call. Maximum number of connections allowed on the incoming queue at any given time is represented by **backlog.** On the server all incoming connections requests from clients come and wait in one special **queue** (from this queue server choose the connection and accept the request). We have to define the limit on queue that how many connections can wait in a queue. Generally this limit is set to 20. But you can set it with any other number also, for example,

listen (socket, 5), call will inform the operating system that server can only allow five client sockets to connect at any one given time.

If the queue is full, then the new connection request will be rejected, which will result in an error in the client application. Queued connections are removed from the queue and completed with the accept() function, which also creates a new socket for communicating with the client.

Example:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVERPORT 1729

int main()
{
  int sockfd;
  struct sockaddr_in ser_addr;

  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    perror("socket error");
```

```
  exit(1);
}

ser_addr.sin_family      = AF_INET;
ser_addr.sin_port        = htons(SERVERPORT);
ser_addr.sin_addr.s_addr = inet_addr(INADDR_ANY);
memset(&(ser_addr.sin_zero), '\0', 8);   /* zero the rest of the struct */

if ( bind(sockfd, (struct sockaddr *)& ser_addr, sizeof(struct sockaddr)) < 0 )
{
  perror("bind");
  exit(1);
}
if ( listen(sockfd, 5) < 0 ) / * Inform the operating sytem that server cn allow
only 5 clients*/
{
  perror("listen error");
  exit(1);
}
  .
  .
  .
 }
```

2.2.5 Accept () System Call

After listen system call accept() completes the process of establishing connection between the server and the client. Remember accept system call is used for stream sockets server like TCP server. It removes the first waiting connection request from the queue of pending connections, creates a new socket with the same properties of old socket (which you specified in accept call) and allocates a new file descriptor for the socket. But think what will happen if there is no pending connection waiting in the queue?

In this case we can have two possibilities one *socket is marked as non-blocking* another *socket is not marked as non-blocking.* In this first case **accept()** blocks the caller until a connection is present and if the socket is marked non-blocking and no pending connections are present on the queue, **accept()** returns an error.

Let's see the syntax of accept() system call

```
#include "sys/types.h"
#include "sys/socket.h"

int accept(int socket, struct sockaddr *clientAddress, int *addressLength);

On success, accept() returns the new socket descriptor.
On failure, it returns -1 and the error is in errno.
```

Here in the code accept() takes three arguments first one is **socket** which represent socket on which server is listening the requests, that's why this is also called as **Listening –socket**. Socket on which the server has successfully completed socket(), bind(), and listen() system call. Second is ***clientAddress** which point to an address (struct sockaddr_in). We can use this address to determine the IP address and port of the client. Third and last one ***addressLength** is generally an integer that will contain the actual length of address structure of client (*addressLength should be set to size of (struct sockaddr_in). Accept() returns -1 on error. If it succeeds, it returns a non-

negative integer that is a descriptor for the accepted socket. Let's see one example, which contains the coding showing all the steps we follow till accept().

Example code for accept ().

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVERPORT 1729

int main()
{
 int sockfd, newsock, len;
 struct sockaddr_in ser_addr, client_address;

 if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
 {
   perror("socket error");
   exit(1);
 }
 if ( bind(sockfd, (struct sockaddr *)& ser_addr, sizeof(struct sockaddr)) < 0 )
 {
   perror("bind error");
   exit(1);
 }
 if ( listen(sockfd, 5) < 0 )
 {
   perror("listen error");
   exit(1);
 }
 len = sizeof(client_address);
 while(1)
 {
  if ((newsock = accept ( sockfd, (struct sockaddr *)&cliaddr, &len))<0)
  {
    perror("accept error");
    exit(1);
  }
```

If accept is successful it returns new socket descriptor returned by accept system call which completes all the five elements of 5-tuple associations while the old socket which we passed in accept () contains only three elements of 5 tuple association (except remote address and remote process). As you know most of the server are concurrent in practical situation so all go for new socket automatically as a part of accept () as given in the following code:

Concurrent Service

```
int sockfd, newsockfd;
if ((sockfd = socket(...)) < 0)
        error_handle("socket error");
if (bind(sockfd, ...) < 0)
        error_handle("bind error");
if (listen(sockfd, ...) < 0)
```

```
        error_handle("listen error);
for ( ; ; ) {
        newsockfd=accept(sockfd, ...); /* blocks */
        if (newsockfd < 0)
                error_handle("accept error");
        if (fork() == 0) {
                close(sockfd);                  /* child */
                do_process(newsockfd);          /* process the request */
                exit(0);
        }
        close(newsockfd);               /* parent */
}
```

When a connection request is received and accepted, the process forks, with the child process serving the connection and the parent waiting for another connection request. Let's see the code for iterative server also:

The iterative server

```
int sockfd, newsockfd;
if ((sockfd = socket(...)) < 0)
        error_handle("socket error");
if (bind(sockfd, ...) < 0)
        error_handle("bind error");
if (listen(sockfd, ...) < 0)
        error_handle("listen error);
for ( ; ; ) {
        newsockfd=accept(sockfd, ...); /* blocks */
        if (newsockfd < 0)
                error_handle("accept error");

        do_process(newsockfd);          /* process the request */
        close(newsockfd);
          }
```

Here the server handles the request using the connected socket descriptor, newsockfd. It then terminates the connection with a close and waits for another connection using the original descriptor, sockfd, which still has its remote address and remote process unspecified.

☞ Check Your Progress 1

1) Which of the following is returned on success in socket system call?
   a) Negative integer value
   b) Big integer value
   c) Small integer value
   d) Text "success"

……

……

……

2) Which of the following header file contains the definition of data types?

a) Socket. h

b) type. h

c) type. h

d) data type. H

……

……

……

3) Listen system call is used on the server in the case of

a) Connection–less common

b) Connection-oriented comma

c) Simplex Comma

d) Duplex Comma full

……

……

……

2.3 ELEMENTARY DATA TRANSFER CALLS

Till now we have studied about connection establishment between client and server, let's start the discussion about data transfer between them.

Once a connection is established between sockets, an application program can send and receive data. Sending and receiving data can be done with any one of the several system calls given in this section. The system calls vary according to the amount of data to be transmitted and received and the state of the socket being used to perform the data transfers. The system call pairs (read, write), (send, recv), (sendto, recvfrom) can be used to transfer data (or communicate) on sockets. Let's discuss each pair of system call one by one. The read() system calls allows a process to receive data on a connected socket without receiving the sender's address. The write system calls can be used with a socket that is in a connected state, as the destination of the data is implicitly specified by the connection. The sendto() subroutine allows the process to specify the destination for a message explicitly. The recvfrom() subroutine allow the process to retrieve the incoming message and the sender's address. The use of the above system call varies from protocol to protocol.

read() and write()

read()

This is used to receive data from the remote machine, it assumes that there is already an open connection present between two machines and it is possible only in case of TCP (the connection oriented protocol in TCP/IP).

Prototype syntax for read system call is

```
#include "sys/types.h"
#include "sys/socket.h"

int read(int sockfd, char void *buff, int buff_len );
```

Condition	returns
If successful	Numbers of bytes read
If EOF	0
If Error	-1

Here in the above syntax of read() first *sockfd* is socket descriptor, *buff* is a pointer to the buffer where we can store the data and *buff_len* is length of buffer or capacity of buffer.

As given above on success, read() return the number of bytes read, if error occurs it returns –1 (and *errno* is set appropriately) and if it returns zero that means it read all data in file and EOF (end of file) has come.

write

When we want to a send some data to a process running on remote machine we use write() system call. Write() assumes that connection is already open between sender and receiver machine, that's why this is limited to process using TCP. Write () function attempts to write the specified number of bytes from the specified buffer to the interface buffer specified socket descriptor (sockfd).

```
#include "sys/types.h"
#include "sys/socket.h"

int write(int sockfd, const void *buff, int buff_len );
```

Condition	returns
If successful	Numbers of bytes written
If Error	-1

Here in the above syntax of **write ()** first argument *sockfd* is socket descriptor, buff is a pointer to the buffer from where we can write the data and *buff_len* is length of buffer or capacity of buffer. As given above in figure on success, the numbers of bytes written are returned (where zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately.

send, sendto, recv, and recvfrom

send, sendto, recv, and recvfrom system calls are similar to the standard read and write system calls but these calls require some additional arguemnts like *flag ,*dest_addr, addrlen and *sour_addr as given below :*

Name of additional Argument(s)	Name of system call
flags	send() and recv().
*flag ,*dest_addr, addrlen*	sendto()
*flag ,*sour_addr, *addrlen*	recvfrom()

You can note *addrlen* argument in *sendto* is an integer value while in *recvfrom* it is a pointer to an integer value.You can compare the syntax of these system calls in the

code given below. You can easily note that the first three arguments *sockfd ,*buff* and *nbytes* are similar to first three arguments of read() and write() argument.

```
#include "sys/types.h"
#include "sys/socket.h"
int send(int sockfd, char *buff, int nbytes, int flags);

int recv(int sockfd, char *buff, int nbytes, int flags);

int sendto(int sockfd, char *buff, int nbytes, int flags,
            struct sockaddr *dest_addr, int addrlen);

int recvfrom(int sockfd, char *buff, int nbytes, int flags,
              struct sockaddr sour_addr, int* addrlen);
```

The use of these system calls depends on the protocol used in your socket association.When we use TCP commonly we use *send()* and *recv()*, allthough *sendto()*, *recvfrom()*, and read() and write() can also work. In case of UDP , if you have bound a IP address and Port, you should use *recv()* for reading, and *send()* for sending. If you have not used bind on the UDP socket it is better to use *sendto()* and *recvfrom()*.

The different flag values are defined in the **sys/socket.h** header file. Flag can be defined as a nonzero value if the application program requires one or more of the following flag values:

MSG_OOB	**Sends or receives out-of-band data.**
MSG_PEEK	**Looks at data without reading.**
MSG_DONTROUTE	**Sends data without routing packets**
MSG_MPEG2	**Sends MPEG2 video data blocks**

Send ()

The send() function initiates transmission of a message from the specified socket to its peer. The send() function sends a message only when the socket is connected.

sockfd is socket descriptor, note that socket must be in connected state before sending data, **buff* is a pointer to data to be transmitted, *nbytes* indicates number of bytes to be sent and *flags toolbars* control flags for special protocol features; set to 0 in most cases. On success, send() returns the number of bytes actually sent.On failure, it returns -1 and the errno.

The following code example shows how you can send data to a connected socket.

```
int i;
  /*  sockfd is the connected socket file descriptor  */
  i = send( sockfd, "Hello World!\n", 13, 0);
  if( i > 0 ){
     printf("sent %d bytes\n", i);
  }
```

sendto()

int sendto(int sockfd, char *buff, int nbytes, int flags, struct sockaddr *dest_addr, int addrlen);

The sendto() function sends a message through a connection oriented or connectionless socket. If sockfd specfic is a connectionless socket, the message is

sent to the address specified by *dest_addr. If sockfd specifies is a connection oriented socket, *dest_addr and addrlen parameters are ignored.

*dest_addr indicate destination address for data to busiest, addrlen represent length of destination address structure and other arguments are similar to send() function explained above. On success, sendto() returns the number of bytes sent. On failure, it returns -1 and the error is in errno.

recv()

int recv(int sockfd, char *buff, int nbytes, int flags);

The recv() function receives a message from a socket. The recv() call can be used on a connection oriented socket and, connectionless socket. If no messages are available at the socket, the recv() call waits for a message to arrive if the socket is blocking. If a socket is nonblocking, -1 is returned and the external variable errno is set.

The flags parameter can be set to MSG_PEEK, MSG_OOB, both, or zero. If it is set to MSG_PEEK, any data returned to the user still is treated as if it had not been read, i.e., the next recv() re-reads the same data.

If successful, recv() returns the number of bytes received, otherwise, it returns -1 and sets errno to indicate the error. recv() returns 0 if the socket is blocking and the connection to the remote node failed.

The following code example shows how you would use recv on a connected socket.

```
int i;
char buff[ 250 ];
/*  clear out buff  */
memset( buff, 0, sizeof( buff ) );
/*  sockfd is the connected socket file descriptor  */
i = recv( sockfd, buff, sizeof(buff), 0);
if( i < 0 )
        {
           printf("error in recving data %d\n", i);
        }
        if( i == 0 )
        {
           printf("socket closed remotely\n");
        }
        if( i > 0 )
        {
    printf("received %d bytes\n", i);
    printf("data :\n%s", buff);
}
```

recvfrom ()

int recvfrom(int sockfd, char *buff, int nbytes, int flags, struct sockaddr *from, int *addrlen);

The recvfrom() system call receives a message from a socket and capture the address from which the data was sent. Unlike the recv() call, which can only be used on a connected stream socket or bound datagram socket, recvfrom() can be used to receive data on a socket whether or not it is connected. If no messages are available at the socket, the recvfrom() call waits for a message to arrive if the socket is blocking. If a socket is nonblocking, -1 is returned and the external variable errno is set.

2.4 CLOSING A SOCKET

There are two ways to close a socket . First is **close(sockfd)** and another is **shutdown(socked, prio).** Let's check what is the difference between both. In close() you can close the socket imeediately, while in shutdown() you can close the socket, after flushing buffers. You can indicate what buffers shutdown should flush with an integer **prio**. The following example(s) show how to use shutdown, and close, on a connected socket.

```
/*  close socket imeediately */
  • close( sockfd );
  • shutdown( sockfd, prio);
/*  close socket, and dont recieve any more */
shutdown( sockfd, 0);
/*  close socket, and dont send any more */
shutdown( sockfd, 1);
/*  close socket, and dont recieve, or send any more */
shutdown( sockfd, 2);
```

close(): The close() function is used to delete a socket descriptor created by the socket() function.

```
#include <socket.h>
#include <uio.h>
int close (socket )
int socket;
```

close() deletes the socket descriptor, sockfd, from the internal descriptor table maintained for the application program and terminates the existence of the communications endpoint. If the socket was connected, the connection is terminated.

shutdown() : The shutdown() function is used to gracefully shut down a socket.

```
#include <socket.h>
#include <uio.h>
int shutdown (sockfd, prio)
int sockfd, prio;
```

The input path can be shut down while continuing to send data, the output path can be shut down while continuing to receive data, or the socket can be shut down in both directions at once but the data queued for transmission is not lost in shutdown(). If prio is 0, further receives are disallowed. If prio is 1, further sends are disallowed. If how is 2, further sends and receives are disallowed.

☞ Check Your Progress 2

1) When read () system calls returns 0 (zero) it means?

   a) Error has occurred

   b) Buffer is empty

   c) Buffer is full

   d) End of file has occurred

   ……

2) Which of the following header file contains the definition of data types?

   a) Disconnected
   b) Ended
   c) Listing
   d) Connected

   ……

   ……

   ……

3) The recvfrom() function receives a message from socket and capture the

   a) Message from pear socket
   b) Address from which the data was sent
   c) Size of buffer from which the data was sent
   d) Port number from the pear socket.

   ……

   ……

   ……

2.5 TCP AND UDP ARCHITECTURES

UDP architectures

As you know broadly the services in TCP/IP are divided in connection oriented and connection-less services, for which TCP and UDP protocols are responsible. It is important for you to understand the architecture of these UDP and TCP communication.

UDP Client algorithm

To understand the communication architecture between UDP client and server, let use different socket calls.

Here we have explained you the step of UDP client algorithm, first you create a socket, then bind it to a local port remember if bind is not used, the kernel will select a free local port), establish the address of the server, write and read from it, and then terminate. In case, client is not interested in a giving reply then there is no need to use bind.

UDP server algorithm

These are steps generally involved in UDP server, here first you create a socket, bind it to a local port, accept and reply to messages from client, and if you want terminate.

The UDP client/server architecture

The system calls are different for a client-server using a connectionless protocol, from the connection-oriented case. The diagram given below in *Figure 2* shows a typical sequence of systems calls for both the client and the server in connection-less (UDP) communication:

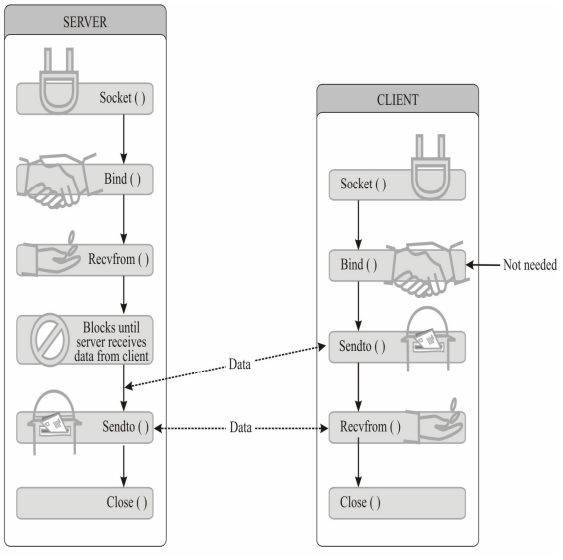


Figure 2: UDP Architecture

TCP architectures

Similarly to understand the architectural communication between TCP client and server we have different socket call in the following section.

TCP Client Algorithm

Here the sequence of steps for connection-oriented client are given. First you will create a socket, bind it to a local port (we usually do not call bind), establish the address of the server, communicate with it, if you want, terminate.

TCP Server Algorithm

These are algorithm sequence for connection oriented server, in this case you first create a socket, bind it to a local port, set up service with indication of maximum number of concurrent services, accept requests from connection oriented clients, receive messages and reply to them and then, finally terminate.

TCP Client/Server Architecture

Typical sequence of system calls to implement TCP clients and servers are given below in the *Figure 3*.

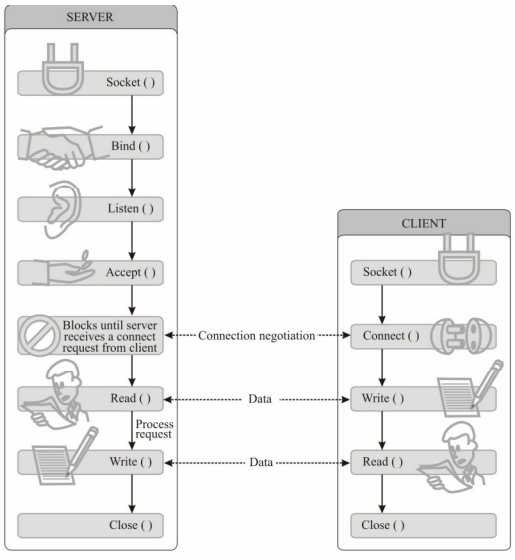


Figure 3:TCP Architecture

2.6 NETWORKING EXAMPLE

An example of an UDP echo client and server is given in this section. The client code is as follows:

UDP echo client

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
```

Include all the necessary header files.

```
#define LINES 128
#define PORT 7200

int main(int argc, char *argv[])
{
 int sockfd;
 int bytesent;
 int byterec;
 struct sockaddr_in ser_addr;
 struct sockaddr_in echo_addr
 socklen_t len= sizeof(ser_addr);
 char sendline[LINES];
 char recvline[LINES + 1];
 if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
 {
  perror("socket error");
  exit(1);
 }
 /* fill address completely with 0's */
 memset(& ser_addr, '\0', sizeof(ser_addr));
 ser_addr.sin_family = AF_INET;
 ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
 ser_addr.sin_port = htons(PORT); ser_addr

 while (1)
 {
  printf("==>");
  fflush(stdout);
  fgets(sendline, LINES, stdin);
  if (strcmp(sendline, "exit\n") == 0)
  break;
  bytesent = sendto(sockfd, sendline, strlen(sendline), 0,
          (struct sockaddr *) &ser_addr, sizeof(ser_addr));
  if (bytesent == -1)
  {
   perror("sendto error");
   exit(1);
  }
  if (strcmp(sendline, "Terminate!\n") != 0)
  {
   if ((byterec = recvfrom(sockfd, recvline, LINES, 0, & echo_addr, &len)) < 0)
   {
```

```
      perror("recvfrom error");
      exit(1);
     }
     recvline[byterec] = '\0';
     printf(" from server: %s\n", recvline);
    }
    }
   close(sockfd);
    printf("echo client normal end\n");
  return 0;
}
```

UDP echo server

The server code is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>

#define LINES 128
#define PORT  7200
int
main(int argc, char *argv[])
{
  int sockfd
  int byterec;
  struct sockaddr_in ser_addr;
  struct sockaddr_in cli_addr;
  socklen_t len=sizeof(cli_addr);
  char recvline[LINES + 1];

   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
  {
    perror("socket error");
    exit(1);
  }
  memset(& ser_addr, '\0', sizeof(ser_addr));
  ser_addr.sin_family = AF_INET;
  ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  ser_addr.sin_port = htons(PORT);
  bind(sockfd, (struct sockaddr *) & ser_addr, sizeof(ser_addr));
  while (1)
  {
   if ( (numrec = recvfrom(sockfd, recvline, LINES, 0,
                (struct ser_addr*) & cli_addr,& len)) < 0)
    {
      perror("recvfrom error");
      exit(1);
    }
    fprintf(stdout, "connection from %s, port %d. Received %d bytes.\n",
          inet_ntoa(cli_addr.sin_addr),
          ntohs(cli_addr.sin_port),
          byterec);
```

```
    fflush(stdout);
    recvline[byterec]='\0';
    if (strcmp(recvline, "Terminator!\n") == 0)
    {
      fprintf(stdout, "a client want me to terminate\n");
      fflush(stdout);
      break;
    }
     if (sendto(sockfd, recvline, byterec, 0, &cli_addr, len) < 0)
    {
      perror("sendto error");
      exit(1);
    }
  }
  fprintf(stdout, "server normal end\n");
  fflush(stdout);
  return 0;
}
```

Check Your Progress 3

1) Draw the sequence of system calls required for implementing TCP Clients and server.

……

……

……

……

……

……

……

2) Draw the sequence of system calls required for implementing UDP clients and server.

……

……

……

……

……

……

……

2.7 SUMMARY

We have discussed different elementary system call and their uses in connection establishment, termination and data transfer in TCP and UDP client server architecture. This unit describes the sockets programming interface. The special needs of network communication are discussed in general terms, and socket types and addressing schemes are introduced. In the first section we have covered the elementary system call to provide you basic knowledge about socket programming. The system calls for data transfer were discussed in detail with an example. We have

also covered the different techniques to close the socket, which will definitely help you during programming. The unit concludes with annotated algorithms of client and server communication programs. In the next unit *Socket Programming* we have given advanced socket calls, which will provide you the in-depth knowledge of the functions and their characteristics that are required for developing network applications.

2.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) C

2) C

3) B

Check Your Progress 2

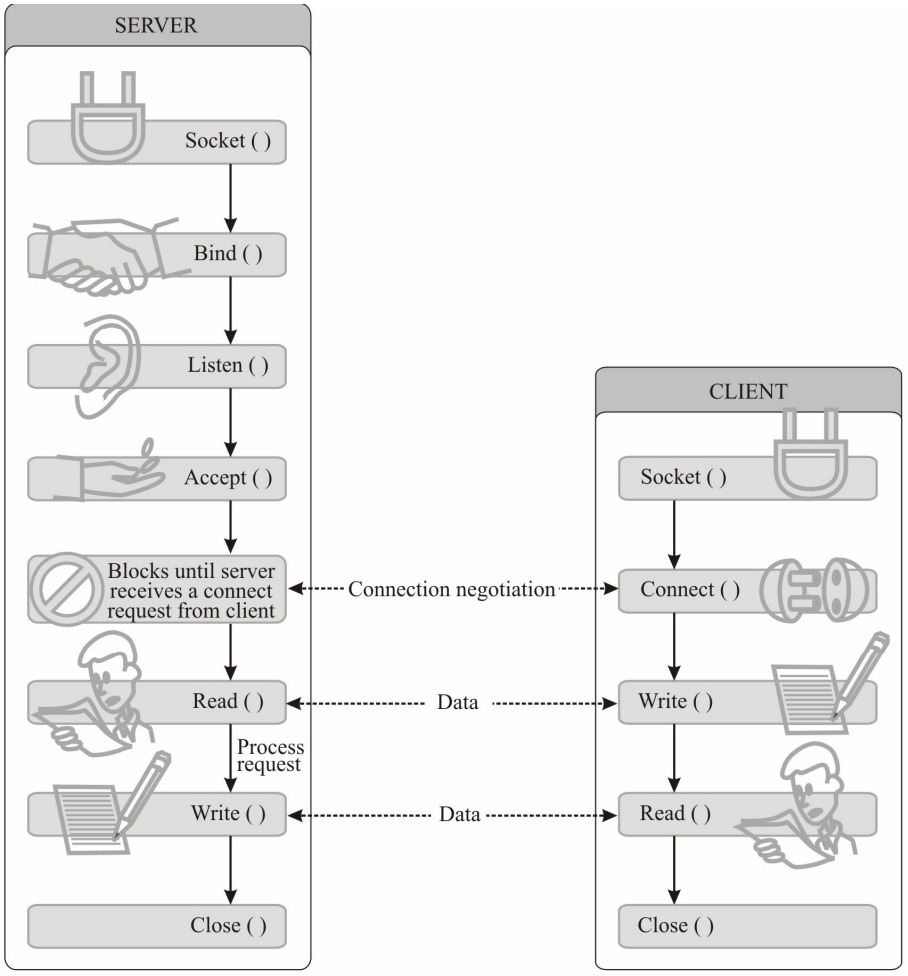
1) D

2) D

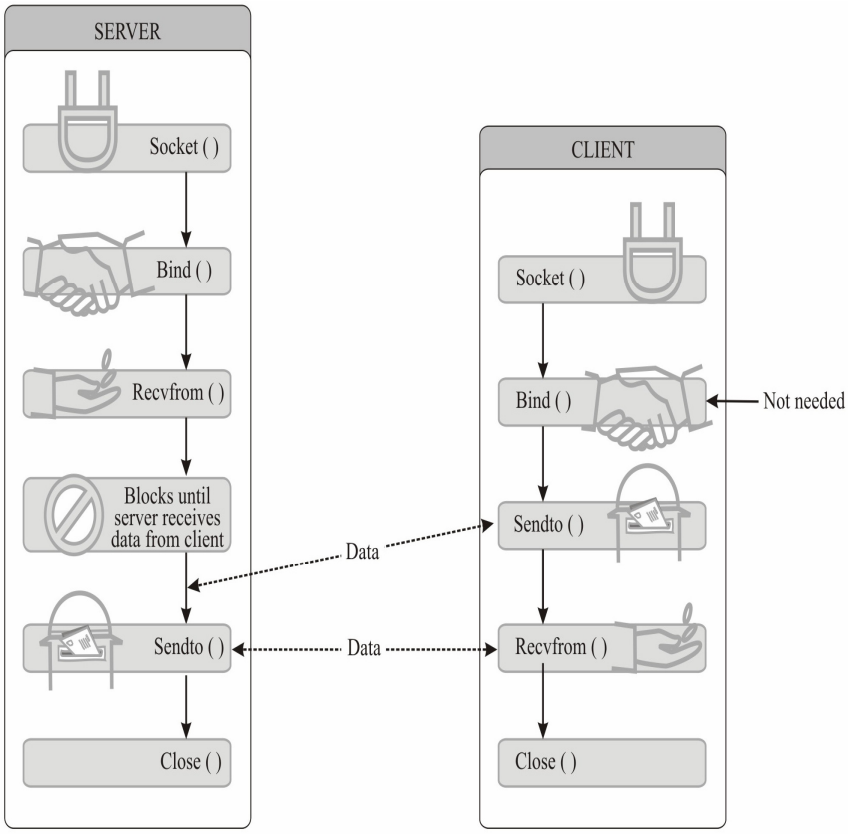
3) B

Check Your Progress 3

1)



2)



2.9 FURTHER READINGS

1) Andrew S. Tanenbaum, *Computer Networks*, Third edition.

2) Behrouz A. Forouzan, *Data Communications and Networking*, Third edition.

3) Douglas E. Comer, *Internetworking with TCP/IP* Vol.1: Principles, Protocols, and Architecture (4th Edition).

4) William Stallings, *Data and Computer Communications*, Seventh Edition.

5) W. Richard Stevens, *The Protocols (TCP/IP Illustrated*, Volume 1).

6) W. Richard Stevens, "*UNIX Network Programming"*, Prentice Hall.

UNIT 3 SOCKET PROGRAMMING

Structure **Page Nos.**

3.0 Introduction 46

3.1 Objectives 46

3.2 Advance System call 47

3.2.1 Data Transfer 47

3.2.2 Byte Operations and Addressing 48

3.2.3 Socket Options 55

3.2.4 Select System Call 57

3.3 Raw Socket 58

3.4 Multiple Recipients 60

3.4.1 Unicasting 60

3.4.2 Broadcasting 60

3.4.3 Multicasting 60

3.5 Quality of Service Issues 61

3.6 Summary 63

3.7 Answers to Check Your Progress 63

3.8 Further Readings 64

3.0 INTRODUCTION

In the previous unit, we have discussed different elementary system calls required for connection establishment, termination and data transfer in TCP and UDP client server architecture. But because of the complexity and the requirements of network communication like, byte ordering problem, addressing schemes, multiple recipients communication and the need of personal protocols development we need many more system calls for network communication that we will discuss in this unit.

3.1 OBJECTIVES

Our objective is to introduce you the advanced socket calls, which will provide you the in-depth knowledge of the functions and their characteristics that are required for developing network applications. After successful completion of this unit, you should be able to:

- have a reasonable understanding of the Advance System calls;
- understand the use of advance data transfer calls;
- describe the Byte Operations and Addressing functions;
- know the use and applications of socket options;
- understand the concept of raw sockets;
- demonstrate an understanding of the network programs for multiple recipients, and
- know the Quality of Service issues in TCP/IP.

3.2 ADVANCE SYSTEM CALL

This section will describe the syntax and semantics of some of the important system calls, organised by their functionalities. The syntax of these calls are given in C language, where the meaning and use of each system call is defined. Then the syntax with necessary parameters and header files are given. They usually return a negative value to indicate an error and a non-negative return value to indicate success. .

3.2.1 Data Transfer

readv(): The readv() function is used to input data from a socket in scatter mode when the input is to be noncontiguous.

```
#include <socket.h>
#include <uio.h>
int readv ( sockfd, buff, buffcnt)
int sockfd;
struct buffec *buff;
int buffcnt;
```

```
The buffec structure is defined as:
struct buffec
{
   caddr_t    buff_base;
   int        buff_len;
};
```

The readv() function can be called with either a socket or file descriptor. The readv() function is same as read(), but scatters the input data into the **buffcnt** buffers specified by the members of the buff array: buff[0], buff[1], ..., buff[buffcnt-1]. Each **buffec** entry specifies the *base address* and *length* of an area in memory where data should be placed. readv() function returns the total number of bytes read on success, 0 is returned if an end-of-file condition exits, otherwise, the value -1 is returned with error code.

writev(): The writev() function is used to output data from a socket in gather mode when the data are not contiguous.

```
#include <socket.h>
#include <uio.h>
int writev ( sockfd, buff, buffcnt )
int sockfd;
struct buffec *buff;
int iovcnt;
```

writev() is also as write(), but gathers the output data from the buffcnt buffers specified by the members of the buff array: buff [0], buff [1], .., buff [buffcnt-1]. writev() function always writes a complete area before proceeding to the next buff entry. It returns the total number of bytes actually written on success, otherwise the value -1 is returned with error code.

recvmsg() : The recvmsg() is used to receive incoming data that has been queued for a connected or unconnected sockets.

```
#include <socket.h>
#include <uio.h>
int recvmsg ( sockfd, messg, flags )
int sockfd;
struct messghdr messg [ ];
int flags;
```

```
struct messghdr
{  caddr_t          messg_name;
   int              messg_namelen;
   struct buffec   *messg_buff;
   int              messg_bufflen;
   caddr_t          messg_accrights;
   int
messg_accrightslen;
};
```

Data is received in "scatter" mode and placed into noncontiguous buffers. The argument **messg** is the pointer to an object of byte structure, messghdr, used to minimize the number of directly supplied parameters. Here messg_name and messg_namelen specify the source address if the socket is unconnected. The messg_buff and messg_bufflen describe the "scatter" locations, as described in read(). A buffer to receive any access rights sent along with the message is specified in messg_accrights, which has length messg_accrightslen. It returns the number of bytes received if successful. 0 is returned if an end-of-file condition exits, otherwise, the value -1 is returned with error code.

```
#include <socket.h>
#include <uio.h>
int sendmsg ( sockfd, messg, flags )
int sockfd;
struct messghdr messg [ ];
int flags;
```

sendmsg(): It is used to send outgoing data on a connected or unconnected socket.

Data is sent in gather mode from a list of noncontiguous buffers. The argument messg is a pointer to an object of byte structure messghdr. This structure is Sam as defined in recvmsg() above. If successful, it returns the number of bytes sent. Otherwise, the value -1 is returned with error code.

3.2.2 Byte Operations and Addressing

Byte Order Conversions: Different types of computers can store data in different ways. A common data format was decided upon for communication between hosts via TCP/IP. The following functions convert data from the host format to the network format, or *vice-versa*. Before going into the details of the data conversion first we can understand the use of these functions.

Function Name	Descriptions
htons	**"**host to network, short", for converting a two-byte integer from host order format to network order
htonl	"host to network, long", for converting a four-byte integer from host order to network order
ntohs	"network to host, short", for converting a two-byte integer from network order to host order
ntohl	"network to host, long", for converting a four-byte integer from network order to host order.

Note the pattern of the function names.

Operation	From format	"to"	To format	Data length
htonl	**h**ost	to	**n**etwork	**l**ong(4byte)
htons	**h**ost	to	**n**etwork	**s**hort(2byte)
ntohl	**n**etwork	to	**h**ost	**l**ong(4byte)
ntohs	**n**etwork	to	**h**ost	**s**hort(2byte)

There are two type of byte ordering:

- Big endian / host byte ordering: most significant bit first.
- Little endian / Network byte ordering: least significant bit first

In the first form, also called 'Big Endian', The Most Significant Byte (MSB) is kept in the lower address, while the Least significant Byte (LSB) is kept in the higher address. In the second form, also called 'Little Endian', the MSB is kept in the higher address, while the LSB is kept in the lower address. As we know (recall unit 5) different computers use different byte ordering (or different endianess), usually depending on the type of CPU they have. The same problem arises when using a long integer: which word (2 bytes) should be kept first in memory? the least significant word, or the most significant word?

In a network protocol, however, there must be a predetermined byte and word ordering. The IP protocol defines what is called 'the network byte order', which must be kept on all packets sent across the Internet. The programmer on a Unix machine is not saved from having to deal with this kind of information. We'll see how the translation of byte orders is solved when we get down to programming.

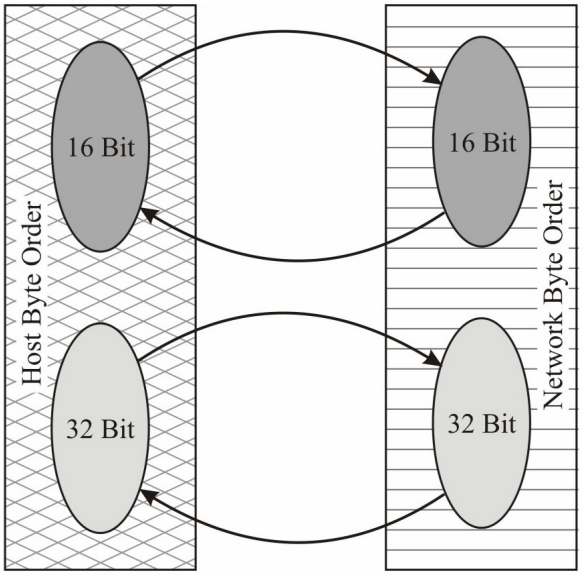


Figure 1: Byte Order Conversion

Here we will take one example to show you how to convert an integer before sending it through a socket and after reading how it can be coverted back into the original form:

1) J=htonl(i); /* converting a four-byte integer from host order to network order */
2) write_data(sockfd, &i, sizeof(i)); /* write data on the socket sockfd*/
3) read_data(sockfd, &i, sizeof(i)); /* read the data*/
4) i=ntohl(i); /* convert a four-byte integer from network order to host order */

If your host order is different from network order and you fail to compensate for it, then your sockets will not work. The sockets API provides a number of conversion functions, including ones that will convert host-order values into network order and vice versa. The details of these functions for converting byte order are given in the following section:

htonl() : It converts 32-bit quantities from host byte order to network byte order.

```
#include <sys/types.h>
#include <inet.h>
unsigned long htonl ( hostlong );
unsigned long hostlong;
```

The htonl() function is provided to maintain compatibility with application programs ported from other systems that byte-swap memory. These systems store 32-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols. If successful, returns the converted value.

htons() : It converts 16-bit quantities from host byte order to network byte order.

```
#include <sys/types.h>
#include <inet.h>
unsigned short htons ( hostshort );
unsigned short hostshort;
```

The htons() function is provided to maintain compatibility with application programs ported from other systems that byte-swap memory. These systems store 16-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols. If successful, returns the converted value.

ntohl() : It converts 32-bit quantities from network byte order to host byte order.

```
#include <sys/types.h>
#include <inet/in.h>
unsigned long ntohl( netlong );
unsigned long netlong;
```

ntohl is portable to other environments, including most UNIX systems, that implement BSD sockets. If successful, returns the converted value.

ntohs() : It converts 16-bit quantities from network byte order to host byte order.

```
#include <sys/types.h>
#include <inet/in.h>
unsigned short ntohs(netshort);
unsigned short netshort;
```

Ntohs () is portable to other environments, including most UNIX systems, that implement BSD sockets. If successful, returns the converted value.

Address Conversion

An Internet address is usually written and specified in the dotted-decimal notation. Internally it becomes part of a structure of type in_addr. To convert between these two representations functions are available. In the following section we will discuss these address conversion function in details.

- inet_aton: ASCII to number (string to binary)
- inet_ntoa: number to ASCII (binary to string)
- inet_addr: like inet_aton.

inet() : The inet functions are a set of routines that construct Internet addresses or break Internet addresses down into their component parts. Routines that convert

between the binary and ASCII ("dot" notation) form of Internet addresses are included. Here, in the given sections, these functions are briefly defined.
unsigned long inet_addr (cp) : inet_addr() interpret character strings (char *cp) representing numbers expressed in the Internet standard dotted (".") notation, returning numbers suitable for use as Internet addresses.

unsigned long inet_network (cp) : inet_network() interpret character strings representing numbers expressed in the Internet standard dotted (".") notation, returning numbers suitable for use as network numbers**.**

char inet_ntoa () : inet_ntoa() takes an Internet address and returns an ASCII string representing the address in dot notation.

unsigned long inet_makeaddr (net, lna) : inet_makeaddr() takes an Internet network number(net) and a local network address (lna) or host number and constructs an Internet address from it.

int inet_lnaof () : inet_netof() break apart Internet host addresses, returning the network number **.**

int inet_netof () : inet_lnaof() break apart Internet host addresses, returning the local network**.**

You must remember that all the Internet addresses are returned in network byte order and all network numbers and local address parts are returned as integer values in host byte order. Values specified using the Internet standard dotted notation take one of these forms:

1) **a.b.c.d** → If 4 parts are specified, each part is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.
2) **a.b.c** → In this case the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as 128.net.host.
3) **a.b** → In this case when two parts are specified the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as net.host.
4) **a** → When only one part is given, the value is stored directly in the network address without any byte rearrangement.

The inet_addr() function returns an Internet address if successful, or a value of -1 if the request was unsuccessful. The inet_network() function returns a network number if successful, or a value of -1 if the request was malformed.

inet_aton()

```
#include <types.h>
#include <socket.h>
#include <in.h>
#include <inet.h>
int inet_aton( cp , in)
const char     *cp;
struct in_addr     *in;
```

As you know inet_aton() converts an ASCII representation of an Internet address to its network internet address. inet_aton() interprets a character string representing

numbers expressed in the Internet standard. notation, returning a number suitable for use as an Internet address.

inet_ntoa()

```
#include <types.h>
#include <socket.h>
#include <in.h>
#include <inet.h>
char *inet_ntoa ( in )
const struct in_addr in;
```

The inet_ntoa() converts an Internet network address to its ASCII "d.d.d.d" representation. inet_ntoa() returns a pointer to a string in the base 256 notation "d.d.d.d". Internet addresses are returned in network byte order (bytes ordered from left to right). inet_ntoa() points to a buffer which is overwritten on each call.

Get useful information

There are number of situations when we need to obtain the official name of a host and we should know about its network address or vice versa. Similarly, sometime we need the name of the local host, the port number associated with a service or the name assigned to a socket there may be many more other cases when we want to get some information of client or server using some known information for this purpose. We have lots of get functions in our socket library which you can use accordingly. Here we have given the details of some get* system call description.

gethostbyaddr(): It is used to obtain the official name of a host when its network address is known.

```
#include <netdb.h>
struct hostent *gethostbyaddr ( addr, len, type )
char *addr;
int len, type;
```

A name server is used to resolve the address if one is available; otherwise, the name is obtained (if possible) from a database maintained on the local system. gethostbyaddr() returns a pointer to an object with the structure hostent.

```
struct hostent
{
char  *h_name;
char  **h_aliases;
int  h_addrtype;
int  h_length;
char  **h_addr_list;
}; #define h_addr h_addr_list [0]
```

Parameters:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A zero-terminated array of network addresses for the host.
h_addr	The first address in h_addr_list; this is for backward compatibility.

The gethostbyaddr() function returns a pointer to the hostent structure, if successful. If unsuccessful, a null pointer is returned, and the external integer h_errno is set to indicate the nature of the error.

gethostbyname() : It is used to obtain the network address or list of addresses of a host when its official name is known.

```
#include <netdb.h>
struct hostent *gethostbyname (name)
char *name;
```

A name server is used to resolve the name if one is available; otherwise, the address is obtained (if possible) from a database maintained on the local system. The gethostbyname() function returns a pointer to an object with the structure hostent as defined above. The gethostbyname() function returns a pointer to the hostent structure, if successful. If unsuccessful, a null pointer is returned, and the external integer h_errno is set to indicate the nature of the error.

gethostname() : It is used to obtain the name of the local host.

```
#include <socket.h>
#include <uio.h>
int gethostname ( name, namelen )
char *name;
int namelen;
```

Here name is the *official name* by which other hosts in the communications domain reference it. Generally, a host belonging to multiple communication domains, or connected to multiple networks within a single domain, has one official name by which it is known. The gethostname() function returns the standard host name for the local system. The parameter namelen specifies the size of the name array. If gethostname() is successful, a value of 0 is returned. A return value of -1 indicates an error.

getservbyname() : It is used to obtain the port number associated with a service when its official name is known.

```
#include <netdb.h>
struct servent *getservbyname ( name, proto )
char *name, *proto;
```

This information is acquired from a service database maintained by the local system. getservbyname() returns a pointer to an object with this given structure:

```
struct servent
{
   char     *s_name;
   char     **s_aliases;
   int      s_port;
   char     *s_proto;
};
```

Parameters:

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides.
s_proto	The name of the protocol to use when contacting the service.

The getservbyname() function sequentially searches from the beginning of the network service database until a matching service name is found, or until the end of the database is reached. It returns a pointer to the servent structure, if successful. If unsuccessful, a null pointer is returned.

getsockname() : It obtains the name assigned to a socket, which is the address of the local endpoint and was assigned with a bind() function.

```
#include <socket.h>
#include <uio.h>
int getsockname ( sockfd, name, namelen )
int sockfd;
struct sockaddr *name;
int *namelen;
```

getsockname() returns the current name for the specified socket. The namelen parameter should be initialized to indicate the amount of space pointed to by name. On return, it contains the actual size of the name returned (in bytes). If getsockname() is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer errno indicates the nature of the error.

☞ **Check Your Progress 1**

1) readv() & writev() functions are used when data are?

   a) Synchronous
   b) Asynchronous
   c) Contiguous
   d) Non-Contiguous

   ……

2) Which of the following function is used for converting a two-byte integer from network order to host order?

   a) h tons()
   b) h tonl()
   c) n tons()
   d) n tohl()

   ……

3) When we want to know the official name of a host and its network address is known, which of the following function is used?

   a) getservby name
   b) getsockname()
   c) gethostbyname()
   d) gethostbyaddr()

   ……

3.2.3 Socket Options

It is possible to set and get a number of options on sockets via the *setsockopt* and *getsockopt* system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

setsockopt(sockfd, level, optname, optval, optlen);

and

getsockopt(sockfd, level, optname, optval, optlen);

The parameters to calls are as follows: *sockfd* is the socket on which the option is to be applied. *Level* specifies the protocol layer on which the option is to be applied; The actual option is specified in *optname*. *Optval* and *Optlen* point to the value of the option and the length of the value of the option, respectively. For *getsockopt*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used. An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs described below will perform this task.

```
#include <sys/types.h>
#include <sys/socket.h>
int type, size;
size = sizeof (int);
if (getsockopt(sockfd, SOL_SOCKET, SO_TYPE, (char *)
&type, &size) < 0) {
```

SO_TYPE get the socket option after the getsockopt call, type will be set to the value of the socket type. Because the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

Lets see the details of getsockopt() and setsockopt() system calls in the following section. Then you will understand the above example in more detail:

setsockopt() : The setsockopt() function is used to manipulate options associated with a socket.

```
#include <socket.h>
#include <uio.h>
#include <inet.h>
#include <tcp.h>
int setsockopt ( sockfd, level, optname, optval, optlen )
int sockfd, level, optname;
char *optval;
int optlen;
```

Options can exist at multiple protocol levels; they are always present at the uppermost socket level. Remember when you are manipulating socket options, the level at which the option resides and the name of the option must be specified properly. level is specified as SOL_SOCKET to manipulate options at the socket level. To manipulate options at any other level, the protocol number of the appropriate protocol is supplied.

For explaining it, let's take an example, the case for changing TCP protocol option the following parameters of setsockopt() should be passed as given below:

level	set to the protocol number of TCP.
optval	identify a buffer that contains the option value.
optlen	length of the option value in bytes.
optname	parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation.

Options at other protocol levels vary in format and name. These options are recognized at the socket level. Each can be set with setsockopt() and examined with getsockopt.

There are a number of socket options which either specialise the behaviour of a socket or provide useful information. These options may be set at different protocol levels and are always present at the uppermost "socket" level. These options (some of these options are given below) allow an application program to customize the behaviour and characteristics of a socket to provide the desired effect.

Option	Parameter Type	Parameter Meaning
SO_BROADCAST	int	Non-zero, requests permission to transmit broadcast datagrams (SOCK_DGRAM sockets only).
SO_DONTROUTE	int	Non-zero, requests bypass of normal routing; route based on destination address only.
SO_KEEPALIVE	int	Non-zero, requests periodic transmission of messages, keep alive (protocol-specific).
SO_OOBINLINE	int	Non-zero, requests that out-of-band data be placed into normal data input queue as received.

getsockopt(): The getsockopt() function is used to retrieve options currently associated with a socket.

```
#include <socket.h>
#include <uio.h>
int getsockopt (sockfd, level, optname,
optval, optlen)
int sockfd, level, optname;
char *optval;
int *optlen;
```

When retrieving socket options, the level at which the option resides and the name of the option must be specified. In continuation of the previous example to indicate that an option is to be returned by the TCP protocol the following parameters of getsockopt() should be indicated as given below:

Level	set to the protocol number of TCP.
Optval	identify a buffer in which the value for the requested option is to be returned.
Optlen	It initially contains the size of the buffer pointed to by the optval and modified on return to indicate the actual size of the value returned. If no option value is to be returned, optval can be supplied as 0.
optname	is passed uninterpreted to the appropriate protocol module for interpretation.

3.2.4 Select System Call

The select() system call is used to synchronise processing of several sockets operating in non-blocking mode. When an application calls recv or recvfrom it is blocked until data arrives for that socket. While the incoming data stream is empty program can do some other job or the situation when a program receives data from multiple sockets. The select function call solves this problem by allowing the program to choose all the socket handles to see if they are available for non-blocking reading and writing operations.

```
#include <socket.h>
#include <uio.h>
int select ( nfds, readfds, writefds, exceptfds, timeout )
int nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
int fd;
fd_set fdset;
FD_SET ( fd, &fdset ) /* Includes a particular descriptor fd in fdset. */
FD_CLR ( fd, &fdset ) /* Removes fd from fdset. */
FD_ISSET ( fd, &fdset ) /* non-zero if fd is a member of fdset, zero
otherwise */
FD_ZERO ( &fdset ) /* Initializes a descriptor set fdset to the null set. */
int fd;
fd_set fdset;
```

Sockets that are ready for reading, ready for writing, or have a pending exceptional condition can be selected. If no sockets are ready for processing, the select() function can block indefinitely or wait for a specified period of time (which may be zero) and then return. Select() examines the I/O descriptor sets whose addresses are passed in readfds, Writefds, and exceptfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first nfds descriptors are checked in each set (in other words, the descriptors from 0 through nfds-1 in the descriptor sets are examined). readfds - A pointer to a set of file and socket descriptors that are to be polled for non-blocking reading and writing operations. writefds, exceptfds are same as readfds, except these sets contain the file/socket handles to poll for non-blocking writing operations and error detection. On return, select() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned.

If timeout is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a zero pointer, the select blocks indefinitely. To affect a poll, the timeout argument should be non-zero, pointing to a zero-valued timeval structure.

If successful, select() returns the number of ready descriptors that are contained in the descriptor sets, or 0 if the time limit expires. Otherwise, the value -1 is returned, and the error code.

Example:

```
fd_set fdset;
struct timeval tv;          // sock is an intialized socket handle
tv.tv_sec = 2;
tv.tv_usec = 500000;   // tv now represents 2.5 seconds
FD_ZERO(&fdset);
FD_SET(sock, &fdset); // adds sock to the file descriptor set
/* wait 2.5 seconds for any data to be read from any single socket */
select(sock+1, &fdset, NULL, NULL, &tv);
if (FD_ISSET(sock, &fdset))
recvfrom(s, buffer, buffer_len, 0, &sa, &sa_len);
else
/* do some other work */
```

3.3 RAW SOCKET

Raw sockets are those sockets, which offer the programmer the possibility to have absolute control over the data, being sent or received through the network. They are very useful when someone needs to create his own protocol but can you think what is a need of creating special protocols? You know with raw sockets you can have your own encrypted traffic tunnels which will enhance your security, you can also optimize the voice and video conference protocols, which may increase the quality and the performance of the sessions.

In this section, you will learn the basics of using raw sockets to insert an IP protocol based datagram into the network traffic. You will learn how to build raw socket scanners or how to perform operations that need to send out raw sockets. Basically, you can send any packet at any time, whereas using the interface functions for your systems IP-stack (connect, write, bind, etc.).

The basic concept of low-level sockets is to send a single packet at one time, with all the protocol headers filled in by the program. Unix provides two kinds of sockets that permit direct access to the network.

- SOCK_PACKET
- SOCK_RAW

SOCK_PACKET receives and sends data on the device link layer. This means, the NIC specific header is included in the data that will be written or read. For most networks, this is the ethernet header. Of course, all subsequent protocol headers will also be included in the data.

The socket type we are going to use, is SOCK_RAW, which includes the IP headers and all subsequent protocol headers and data. A standard command to create a datagram socket is:

```
socket (PF_INET, SOCK_RAW, IPPROTO_UDP);
```

The moment it is created, you can send any IP packets over it, and receive an IP packets.

Note that even though the socket is an interface to the IP header, it is transport layer specific. That means, for listening to TCP, UDP and ICMP traffic, you have to create 3 separate raw sockets, using IPPROTO_TCP, IPPROTO_UDP and IPPROTO_ICMP.

With this knowledge, we can, for example, already create a small sniffer, that dumps out the contents of all tcp packets we receive.

As you see, we are skipping the IP and TCP headers which are contained in the packet, and print out the payload, the data of the session/application layer, only.

```
int fd = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);
char buffer[8192]; /* single packets are usually not bigger than 8192 bytes */
while (read (fd, buffer, 8192) > 0)
printf ("Caught tcp packet: %s\n",
buffer+sizeof(struct iphdr)+sizeof(struct tcphdr));
```

To inject your own packets, all you need to know is the structures of the protocols that need to be included. It is recommended to build your packet by using a struct, so you can comfortably fill in the packet headers. Unix systems provide standard structures in the header files (eg. <netinet/ip.h>). You can always create your own structs, as long as the length of each option is correct. Now, by putting together the knowledge about the protocol header structures with some basic C functions, it is easy to construct and send any datagram(s). To make use of raw packets, knowledge of the basic IP stack operations is essential.

☞ Check Your Progress 2

1) How Many parameters are in setsockopt () system call?
   a) three
   b) four
   c) five
   d) two

   ……
   ……
   ……
   ……

2) Which of the following option should be used for by passing the normal routing?
   a) SO_KEEPALIVE
   b) SO_BYPASS
   c) SO_BYPASSROUTER
   d) SO_DONTROUTE

   ……
   ……
   ……
   ……

3) Which of the following sockets has permission to directly access the network?

a) SOCK_STREAM

b) SOCK_DATAGRAM

c) SOCK_PACKET

d) SOCK_TCP

……

……

3.4 MULTIPLE RECIPIENTS

There are different ways of transmitting a message over a network, one way in which a data is sent from a single source to a specified destination is known as unicasting, another way is multicasting in which data is sent to a specified group of destinations and in broadcasting where data is sent to all connected destinations. Let's discuss these transmission ways in detail:

3.4.1 Unicasting

Unicast is the term used to describe communication where a piece of information is sent from one point to another point. In this case there is just one sender, and one receiver. In unicast transmission a packet is sent from a single source to a specified destination, is still the predominant form of transmission on LANs and within the Internet. For example in our earlier sections we have given the implementation of unicasting.

3.4.2 Broadcasting

Broadcast is the term which is very familiar to all of us, and it has a traditional meaning associated with TV and radio. It generally means as a transmission that can be received by everyone having the correct equipment. In this case there is just one sender, but the information is sent to all connected receivers. Broadcast transmission is supported by most of the LANs, for example the ARP (address resolution protocol) uses this to send an address resolution query to all computers on a LAN.

Sending Broadcasting message

To send a broadcast message, first of all a datagram socket should be created as given below:

sockfd = socket(AF_INET, SOCK_DGRAM, 0);

Then the socket option should be set for allowing broadcasting:

```
int b_on = 1;
setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &b_on, sizeof (on));
/*as we have studied earlier in the course*/
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(sockfd, (struct sockaddr *) &sin, sizeof (sin));
```

3.4.3 Multicasting

Multicasting is the term used to describe communication where a piece of information is sent from one point to a set of other points. In this case there is one sender, and the information is distributed to a set of receivers. One example of this we can see in email and chatting groups on the Internet.

IP multicast provides dynamic many-to-many connectivity between a set of senders (at least 1) and a group of receivers. The format of IP multicast packets are identical to that of unicast packets and is distinguished only by the use of a special class of destination address (class D IP address), which denotes a specific multicast group. Most of the LAN's are able to support the multicast transmission mode. It is inherently supported by the shared LANs, because in that case all packets reach all network interface cards connected to the LAN.

Sending multicasting message

To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255. A socket option is available to override the default for subsequent transmissions from a given socket:

struct in_addr addr;

setsockopt(sock, IPPROT O_IP, IP_MULTICAST_IF, &addr, sizeof (addr));

Where "addr" is the local IP address of the desired outgoing interface.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether or not subsequent datagrams are looped back:

u_char loop;

setsockopt(sock, IPPROT O_IP, IP_MULTICAST_LOOP, &loop,

sizeof(loop));

where *loop* is set to 0 to disable loopback, and set to 1 to enable loopback. This option improves performance for applications that may have no more than one instance on a single host (such as a router demon), by eliminating the overhead of receiving their own transmissions.

To receive multicast datagrams sent to a particular port, it is necessary to bind to that local port, leaving the local address unspecified (i.e., INADDR_ANY). To receive multicast datagrams sent to a particular group and port, bind the local port, the local address set to the multicast group address. Once bound to a multicast address, the socket cannot be used for sending data.

More than one process may bind to the same SOCK_DGRAM UDP port or the same multicast group and port if the *bind* call is preceded by:

int on = 1;

setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on));

All processes sharing the port must enable this option. Every incoming multicast or broadcast UDP datagram destined to the shared port is delivered to all sockets bound with the port. For backwards compatibility reasons, this does not apply to incoming unicast datagrams. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port.

3.5 QUALITY OF SERVICE ISSUES

"Quality of service" has become a big buzzword. This term conveys about as much useful information about what the technology offers as being told that it is "high performance".

The Internet is designed on top of the Internet Protocol, a packet-switching technology that is designed to get packets from point "A" to point "B" in whatever

way is most effective, without the user necessarily having any ability to know what route will be taken. In fact, some packets in the same data stream may be sent along different routes. Packets may be stored for a while before being forwarded to their destination, or even dropped and retransmitted.

For most applications, such as simple file or message transfers, this is perfectly fine. However, there are applications where this sort of service is simply of "too low quality". In these cases, the nature of how the data is delivered is more important than merely how fast it is, and there is a need for technologies or protocols that offer "quality of service". This general term can encompass a number of related features; common ones include the following:

- **Bandwidth Reservation**: The ability to reserve a portion of bandwidth in a network or interface for a period of time, so that two devices can count on having that bandwidth for a particular operation. This is used for multimedia applications where data must be streamed in real-time and packet rerouting and retransmission would result in problems. This is also called resource reservation.
- **Latency Management**: A feature that limits the latency in any data transfer between two devices to a known value.
- **Traffic Prioritization**: In conventional networks, "all packets are created equal". A useful QoS feature is the ability to handle packets so that more important connections receive priority over less important one.
- **Traffic Shaping**: This refers to the use of buffers and limits that restrict traffic across a connection to be within a pre-determined maximum.
- **Network Congestion Avoidance**: This QoS feature refers to monitoring particular connections in a network, and rerouting data when a particular part of the network is becoming congested.

So, in essence, quality of service in the networking context is analogous to quality of service in the "real world". Some applications, especially multimedia such as voice, music and video, are time-dependent and require a constant flow of information more than the raw bandwidth.

Key Concept: The generic term quality of service describes the characteristics of how data is transmitted between devices, rather than just how quickly it is sent. Quality of service features seek to provide more predictable streams of data rather than simply faster ones. Examples of such features include bandwidth reservation, latency minimums, traffic prioritization and shaping, and congestion limitation. Quality of service is more important for special applications such as multimedia than for routine applications such as those that transfer files or messages.

To support quality of service requirements, many newer technologies have been developed or enhanced to add quality of service features to them. This includes the ability to support isochronous transmissions, where devices can reserve a specific amount of bandwidth over time to support applications that must send data in real time.

☞ Check Your Progress 3

1) The transmission in which data is sent to a specific group of destination is called.

   a) Unicasting

   a) Multicasting

   b) Broadcasting

   c) All the above

……..……...

……

……

2) Which of the following protocol uses the transmission in local area network?

a) ARP

b) BRP

c) TCP

d) UDP

……..…...…

……

……

3) Which of the following condition is applied to disable the loop back in setsockopt () system call.

a) loop is set to 1

b) loop is set to 0

c) loop is set to 2

d) loop is set to 1

……..…...…

……..…...

3.6 SUMMARY

In this unit we have discussed different practical issues about socket programming. In the first section we have covered the advance system call to provide you enhanced knowledge about socket programming. The address conversion and byte ordering routines were discussed with practical syntax of these routines. To help you for designing your own protocols raw sockets were discussed with their use in network programming. We have also covered the techniques to send data to all the connected computers or some group of computers in a network. In the end of this unit we have compared the services of TCP/IP protocols, which will definitely help you in TCP/IP programming. In the next block *Lab Manual* we have given you some exercises in the lab sessions based on these system calls.

3.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) D

2) C

3) D

Check Your Progress 2

1) C

2) D

3) C

Check Your Progress 3

1) B
2) A
3) B

3.8 FURTHER READINGS

1) Andrew S. Tanenbaum, *Computer Networks*, Third edition.

2) Behrouz A. Forouzan, *Data Communications and Networking*, Third edition.

3) Douglas E. Comer, *Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture* (4th Edition).

4) William Stallings, *Data and Computer Communications*, Seventh Edition.

5) W. Richard Stevens, *The Protocols (TCP/IP Illustrated, Volume 1)*.

6) W. Richard Stevens, *"UNIX Network Programming"*, Prentice Hall.